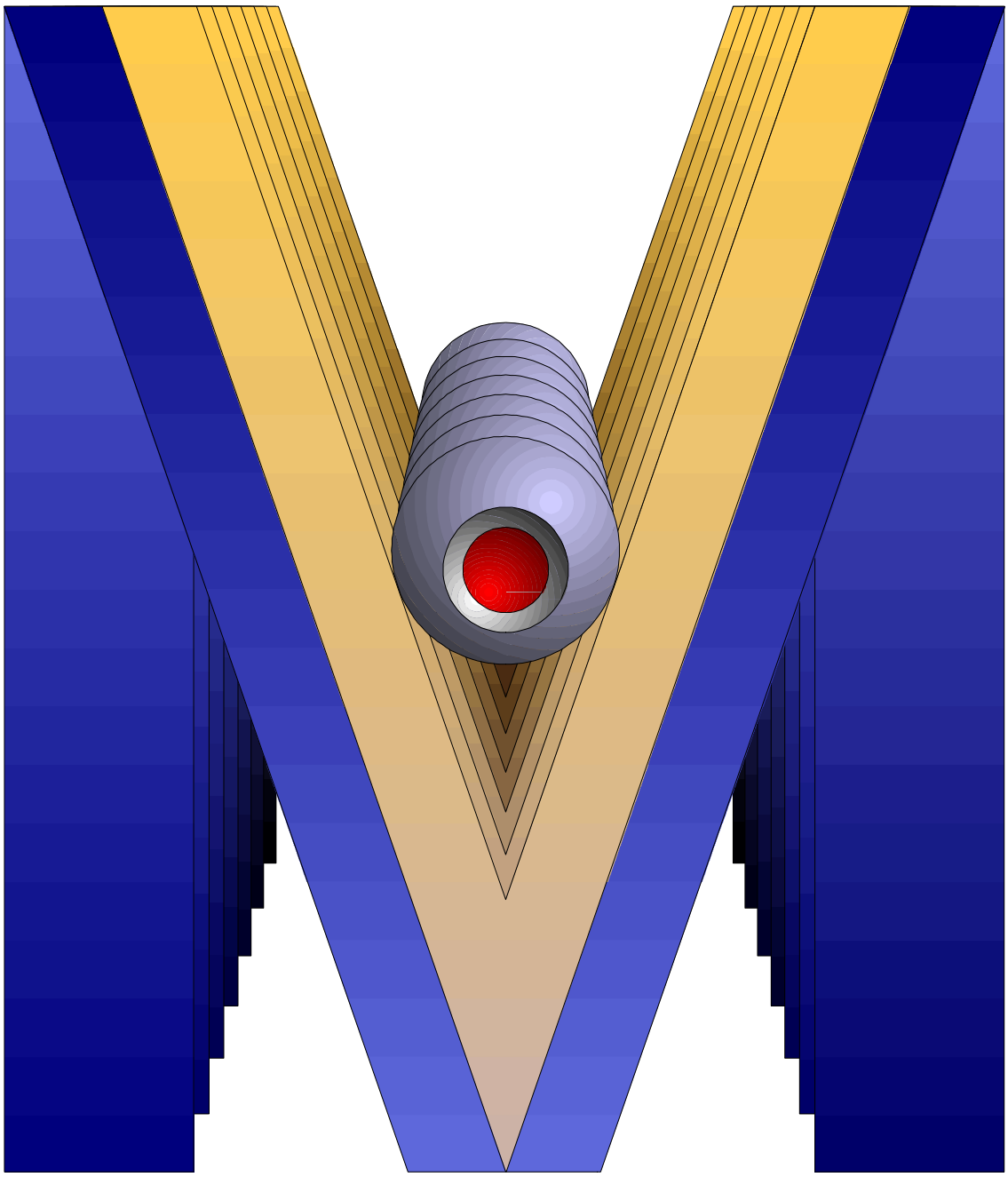
IsoPod™ Users Manual

Virtually Parallel



Machine Architecture

Warranty

New Micros, Inc. warrants its products against defects in materials and workmanship for a period of 90 days. If you discover a defect, New Micros, Inc. will, at its option, repair, replace, or refund the purchase price. Simply call our sales department for an RMA number, write it on the label and return the product with a description of the problem. We will return your product, or its replacement, using the same shipping method used to ship the product to New Micros, Inc. (for instance, if you ship your product via overnight express, we will do the same). This warranty does not apply if the product has been modified or damaged by accident, abuse, or misuse.

Copyrights and Trademarks

Disclaimer of Liability

New Micros, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs of recovering, reprogramming, or reproducing any data stored in or used with New Micros, Inc. products.

Internet Access

Web site: http://www.newmicros.com

This manual: http://www.newmicros.com/store/product_manual/isopod.zip

Email technical questions: nmitech@newmicros.com

Email sales questions: nmisales@newmicros.com

Also see “Manufacturer” information near the end of this manual.

Internet IsoPod™ Discussion List

We maintain the IsoPod™ discussion list on our web site. Members can have all questions and answers forwarded to them. It’s a way to discuss IsoPod™ issues.

To subscribe to the IsoPod™ list, visit the Discussion section of the New Micros, Inc. website.

This manual is valid with the following software and firmware versions:
IsoPod V1.0

If you have any questions about what you need to upgrade your product, please contact New Micros, Inc.

GETTING STARTED

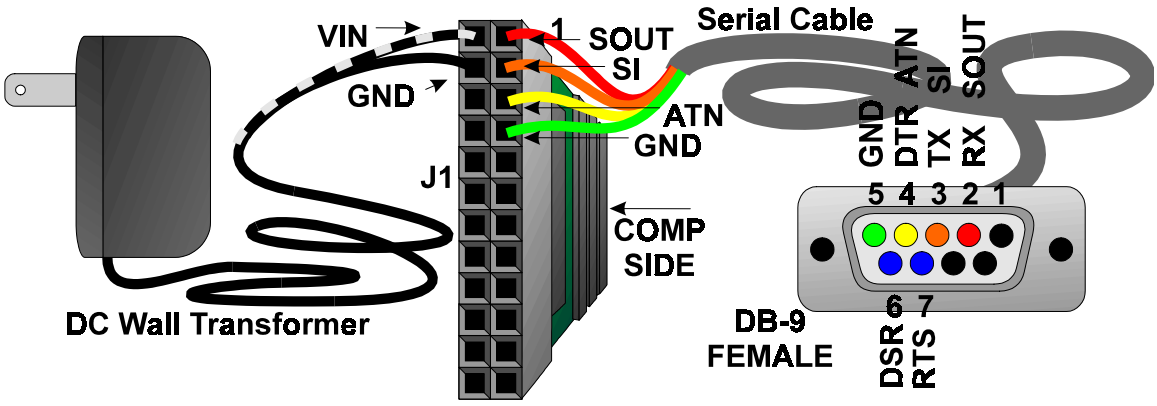
Thank you for buying the IsoPod™. We hope you will find the IsoPod™ to be the incredibly useful small controller board we intended it to be, and easy to use as possible.

If you are new to the IsoPod™, we know you will be in a hurry to see it working.

That's okay. We understand.

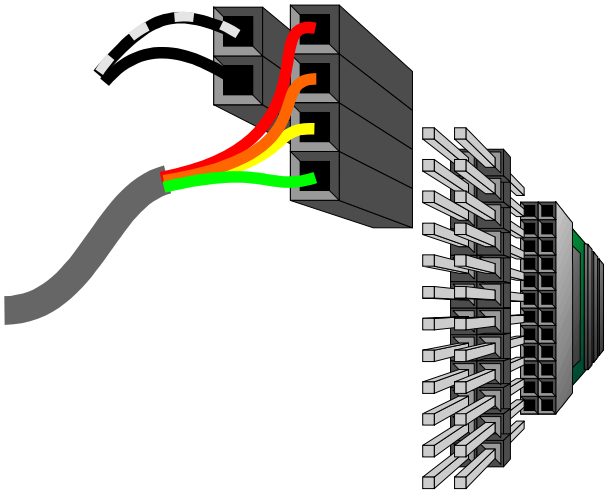
Let's skip the features and the tour and discussion of Virtually Parallel Machine Architecture™ (VPMA) and get right to the operation. Those points can come later. Once we've got communications, then we can make some lights blink and know for sure we're in business. Let's make this "pod" talk to us!

We'll need PC running a terminal program. Then we'll need a serial cable to connect from the PC to the IsoPod™ (which, hopefully, you've already gotten from us). Then we need power, such as from a 6VDC wall transformer (which, hopefully, you've already gotten from us). (If not, you can build your own cable, and supply your own power supply. Instructions are in the back of this manual in Connectors.) If we have those connections correct, we will be able to talk to the IsoPod™ interactively.



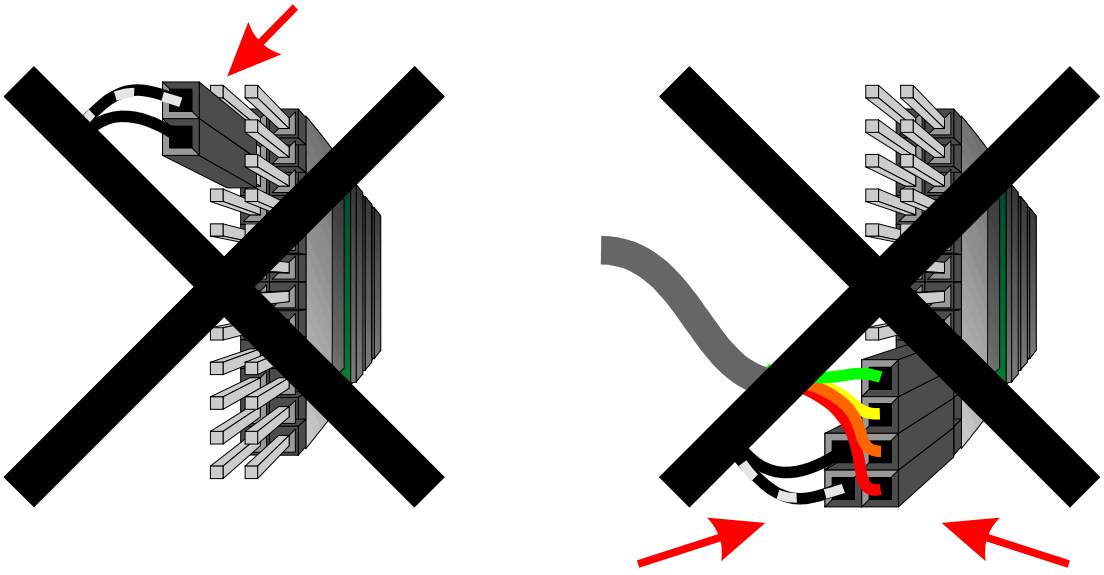
These connections are all made on a few pins of J1, which is a female .1" dual row connector. Download from http://www.newmicros.com/store/product_manual/isopod.zip the manual and read the rest if you haven't yet.

Generally, an intermediate double male header strip will be used to mate from J1 to the Wall transformer single row female connector, and to the Serial Cable single row female connector.



(There are other options we'll discuss later. If you are using your IsoPod™ with our Prototyping Board, these connections will be a little simpler. Follow directions in the Prototyping Board Manual if you are using it.)

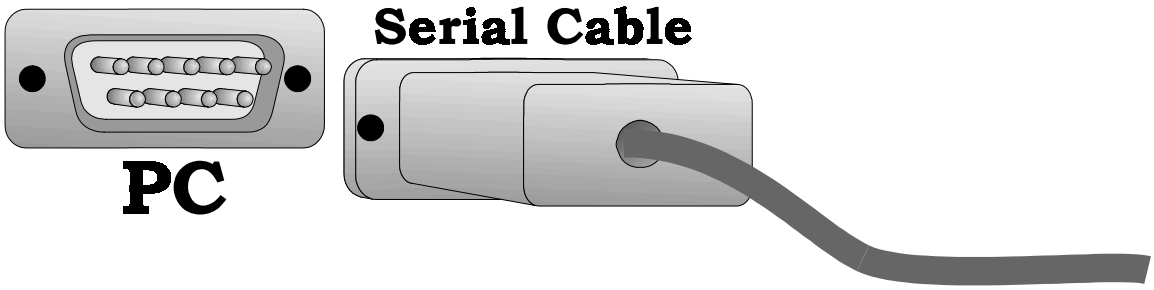
Your chief concern now, is not hooking the serial cable or power cable up on the wrong connector; the wrong pins on the right connector; or backwards or rotated on the right connector. Pay close attention how the connectors go on. There is no protection to prevent plugging in on the .1" dual row headers the wrong way.



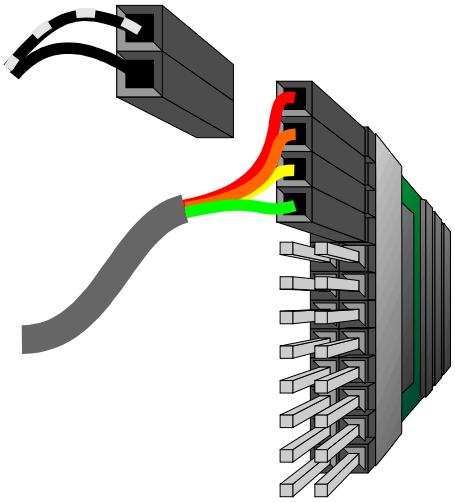
Once you have your serial cable and connectors, and wall transformer and connectors, ready, follow these steps.

Start with the PC: Install and run the MaxTerm program, or, find and start Hyperterm. Set the terminal program for communications channel (COMM1, COMM2, etc.) you wish to use, and set communications settings to (9600 8N1). Operate the program to get past the opening set ups and to the terminal screen, so it is ready to communicate. (If necessary, visit the chapters on MaxTerm and Hyperterm if you have trouble understanding how to accomplish any of this.)

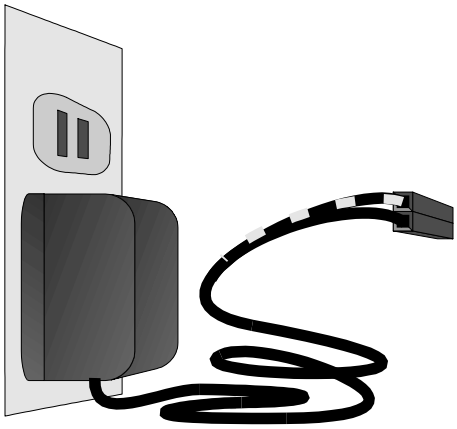
Hook the computer end of the serial cable (usually a DB-9 connector, but may be a DB-25, or other, on older PC's) to the PC's communication channel selected in the terminal program.



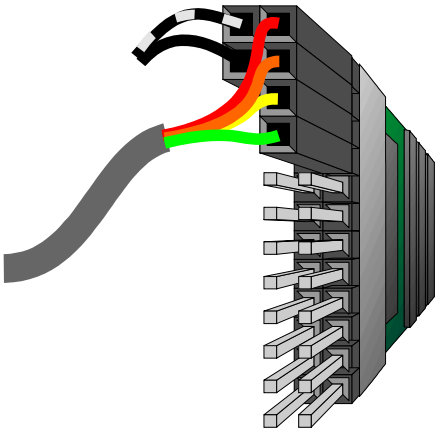
Now hook the IsoPod™ end of the serial cable to the IsoPod™ with connections as shown in the instructions. See the illustration here:



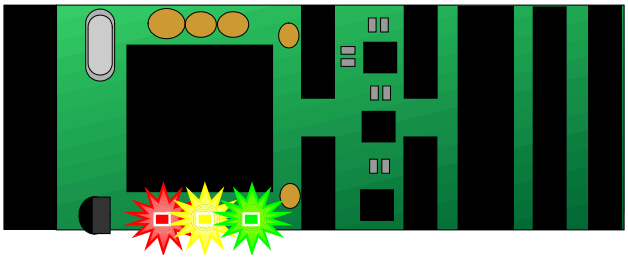
Plug the wall transformer into the wall, but do not plug it into the board yet.



Now, while watching the LED's plug in the wall transformer connector to the power pins on the IsoPod™ board. Be *very* careful not to get a misalignment here, because it will likely kill the board. See the illustration here:



All three LED's should come on. If the LED's do not light, unplug the power to the IsoPod™ quickly.



Now check the screen on the computer. When the power is applied, before any user program installed, the PC terminal program should show "`IsoMax™ V1.0`" (or whatever the version currently is, see upgrade policy later at the end of this chapter).

If the LED's don't light, and the screen doesn't show the message, unplug the power to the IsoPod™. Go back through the instructions again. Check the power connections,

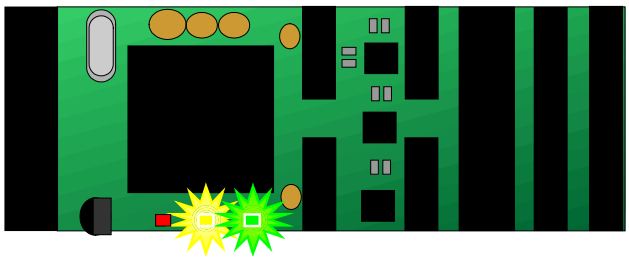
particularly for polarity. (This is the most dangerous error to your board.) If the LED's come on but there is no communication, check the terminal program. Check the serial connections, particularly for a reversal or rotation. Try once more. If you have no success, see the trouble shooting section of this manual and then contact technical support for help, before going further. Do not leave power on the board for more than a few seconds if it does not appear to be operational.

Normally at this point you will see the prompt on the computer screen "`IsoMax™ V1.0`". Odds are you're there. Congratulations! Now let's do something interactive with the IsoPod™.

In the terminal program on the PC, type in, "`WORDS`" (all in "caps" as the language is case sensitive), and then hit "`Enter`". A stream of words in the language should now scroll up the screen. Good, we're making progress. You are now talking interactively with the language in the IsoPod™.

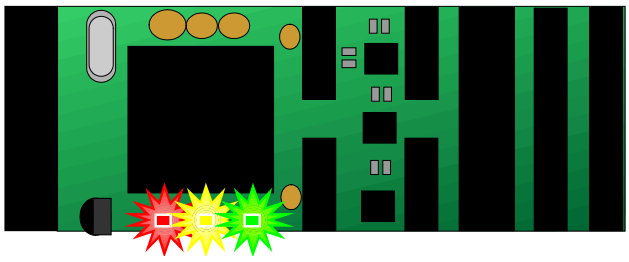
Now let's blink the LED's. Port lines control the LED's. Type:

```
REDLED OFF
```



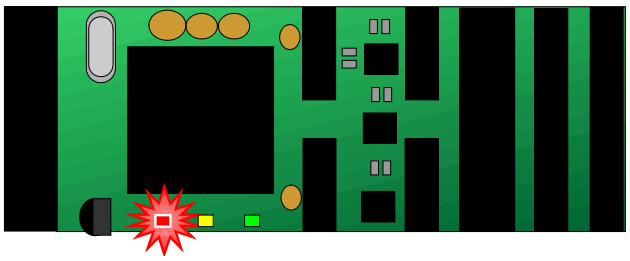
To turn it back on type:

```
REDLED ON
```

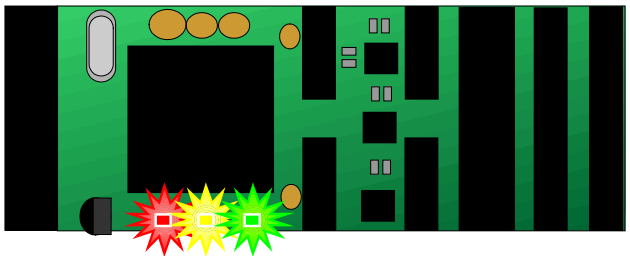


Now let's use the Yellow and Green LED's. Type:

```
YELLED OFF GRNLED OFF
```



```
To turn it back on type:

 YELLED ON GRNLED ON
```



So. Now you should have a good feeling because you can tell your IsoPod™ is working. It's time for an overview of what your IsoPod™ has for features.

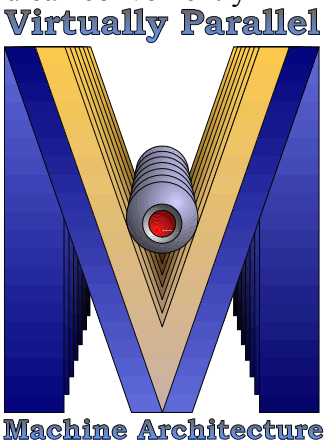
First though, a few comments on IsoMax™ revision level. The first port of IsoMax™ to the IsoPod™ occurred on May 27, 2002. We called this version V0.1, but it never shipped. While the core language was functional as it then was, we really wanted to add many I/O support words. We added a small number of words to identify the port lines and turn them on and off and shipped the first public release on June 3, 2002. This version was V0.2. Currently V0.3 is under development which will have support words for many of the built in hardware functions, and V0.4 is already planned which will had emulation of hardware features on the port lines. As we approach a more complete version, eventually we will release V1.0. We want all our original customers to have the benefit of the extensions we add to the language. Any IsoPod™ purchased prior to V1.0 release can be returned to the factory (at customer's expense for shipping) and we will upgrade the V0.x release to V1.0 without charge.

INTRODUCTION

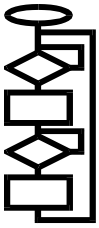
Okay. We should be running. Back to the basics.

What is neat about the IsoPod™? Several things. First it is a very good micro controller. The IsoPod™ was intended to be as small as possible, while still being useable. A careful balance between dense features, and access to connections is made here. Feature density is very high. So secondly, having connectors you can actually "get at" is also a big plus. What is the use of a neat little computer with lots of features, if you can conveniently only use one of those features at a time?

The answer is very important. The neatest thing about the IsoPod™ is software giving Virtually Parallel Machine Architecture!

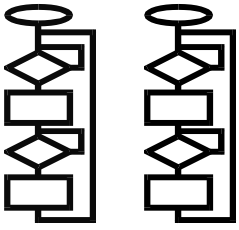


Virtually Parallel Machine Architecture (VPMA) is a new programming paradigm. VPMA allows small, independent machines to be constructed, then added seamlessly to the system. All these installed machines run in a virtually parallel fashion.



In an ordinary high level language, such as C, Basic, Forth or Java, most anyone can make a small computer do one thing well. Programs are written flowing from top to bottom. Flow charts are the preferred diagramming tools for these languages. Any time a program must wait on something, it simply loops in place. Most conventional languages follow the structured procedural programming paradigm. Structured programming enforces this style.

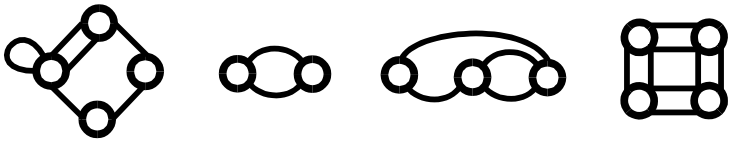
Getting two things done at the same time gets tricky. Add a few more things concurrently competing for processor attention, and most projects start running into serious trouble. Much beyond that, and only the best programmers can weave a program together running many tasks in one application.



Most of us have to resort to a multitasking system. (Windows and Linux are the most obvious examples of multitasking systems.) For a dedicated processor, a multitasking operating system adds a great amount of overhead for each task and an unpleasant amount of program complexity.

The breakthrough in IsoMax™ is the language is inherently "multitasking" without the overhead or complexity of a multitasking operating system. There's really been nothing quite like it before. Anyone can write a few simple machines in IsoMax™ and string them together so they work.

Old constrained ways of thinking must be left behind to get this new level of efficiency. IsoMax™ is therefore not, and cannot be, like a conventional procedural language. Likewise, conventional languages cannot become IsoMax™ like without loosing a number of key features which enforces Structured Programming at the expense of Isostructure.



In IsoMax™, all tasks are handled on the same level, each running like its own separate little machine. (Tasks don't come and go, like they do in multitasking, any more than you'd want your leg to come and go while you're running.) Each machine in the program is like hardware component in a mechanical solution. Parts are installed in place, each associated with their own place and function.

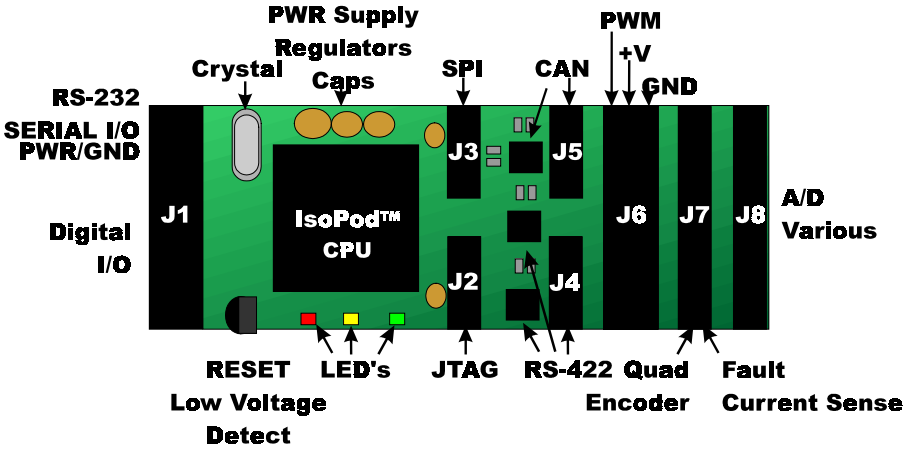
Programming means create a new processor task fashioned as a machine, and debug it interactively in the foreground. When satisfied with performance, you install the new machine in a chain of machines. The machine chain becomes a background feature of the IsoPod™ until you remove it or replace it.

The combination of VPMA software and diverse hardware makes IsoPod™ very versatile. It can be used right side up by J1 with a controller interface board providing an area for prototyping circuitry. It can be used as a stand-alone computer board, deeply embedded inside some project. Perhaps in a mobile robot mounted with double sided sticky tape or tie wraps (although this would be less than a permanent or professional approach to mounting). It can be the controller on a larger PCB board. It can be flipped over and plugged into a carrier board to attach to all signals. A double male right angle connector will convert J1 from a female to a male for such application (however the LED's may no longer be visible) and the mating force of the connectors can sufficiently hold the board in place for most applications. Using a cabled or adapter, it can be plugged into a 24-pin socket of a "stamp-type" controller, to upgrade an existing application.

An IsoPod™ brings an amazing amount power to a very small space, at a very reasonable cost. You'll undoubtedly want to have a few IsoPod™ 's on hand for your future projects.

QUIK TOUR

Start by comparing your board to the diagram below. Most of the important features on the top board are labeled.



The features most important to you will be the connectors. The following list gives a brief description of each connector and the signals involved.

- J1 Serial, Power, General Purpose I/O
- J2 JTAG connector
- J3 SPI
- J4 RS-422/485 Serial Port
- J5 CAN BUS Network Port
- J6 Servo Motor Outputs x 12
- J7 Motor Encoder x 2
- J8 A/D Various

On the left is connector J1. Digital I/O, the power and serial connections are found here. J1 is a female connector. To attach the power and serial connections we need either male pins, or better yet, a male-to-male intermediate header.

All other connectors are dual or triple row male headers. Connection can be made with female headers with crimped wire inserts, or IDC headers with soldered or cabled wires.

Signals were put on separate connectors where possible, such as with the SPI, RS-422, the Can Bus, and PWM connectors. The male headers allow insertion of individually hand-crimped wires in connectors where signals are combined. For instance, R/C Servo motor headers often come in this size connection with a 3x1 header. These can plug directly onto the board side by side on the PWM connector.

To the far left, the low voltage detect and the crystal are just to the right of J1.

The large chip next to them is the CPU.

Three LED's, Red, Yellow and Green, are along the bottom of the CPU, and are dedicated to user control.

Another row of chips between J2/3 and J4/5 are the CAN BUS and RS-422/483 drivers.

On the bottom of the board the largest components are the voltage regulators. If the total current draw were smaller, we could make a smaller supply, but to be sure every user could get enough power to run at full speed, these larger parts were necessary. A smaller module, which will replace the regulators, is also planned.

A few smaller chips are also on the bottom side, the RS-232 transceiver and the LED driver, and a handful of resistors and capacitors.

PROGRAMMING

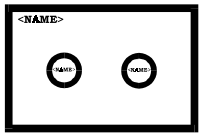
Under construction…

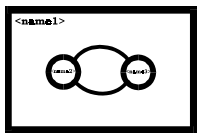
IsoMax is a programming language based on Finite State Machine (FSM) concepts applied to software, with a procedural language (derived from Forth) underneath it. The closest description to the FSM construction type is a "One-Hot" Mealy type of Timer Augmented Finite State Machines. More on these concepts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

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>` `APPEND-STATE <name>` …

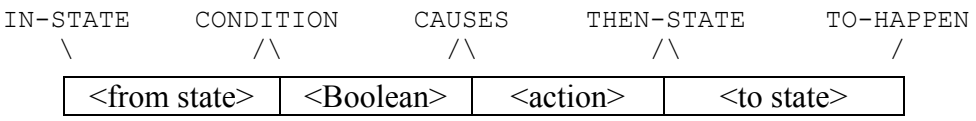
4	Describe transitions from states to states 	`IN-STATE`
		`  <state>`
		`CONDITION`
		`  <Boolean>`
		`CAUSES`
		`  <action>`
		`THEN-STATE`
		`  <state>`
		`TO-HAPPEN`

5	Test and Install	{as required}

What 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.

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 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 by looking at some very simple examples. 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 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. (NON-RETRIGGERABLE ONE-SHOT TIMER: Produces a preset timed output signal on the occurrence of an input signal. The timed output response may begin on either the leading edge or the trailing edge of the input signal. The preset time (in this case: infinity) is independent of the duration of the input signal.) For an example of a hardware non-retriggerable one-shot, see http://www.philipslogic.com/products/hc/pdf/74hc221.pdf.

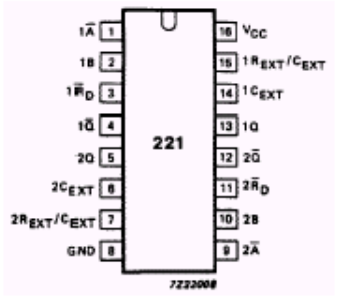


Fig.1 Pin configuration.

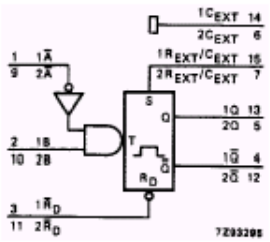


Fig.2 Logic symbol.

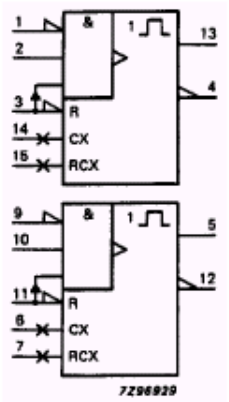
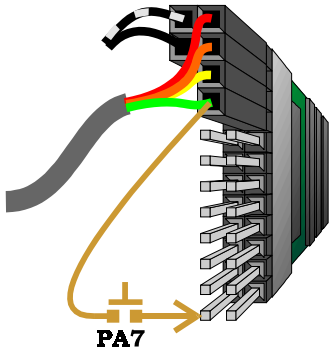


Fig.3 IEC logic symbol.

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.)

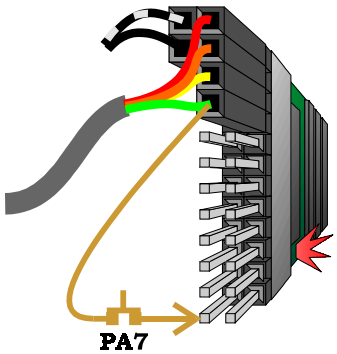
Now if you want, type these lines shown above in. (If you are reading this manual electronically, you should be able to highlight the text on screen and copy the text to the clipboard with Cntl-C. Then you may be able to paste into your terminal program. On MaxTerm, the command to down load the clipboard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

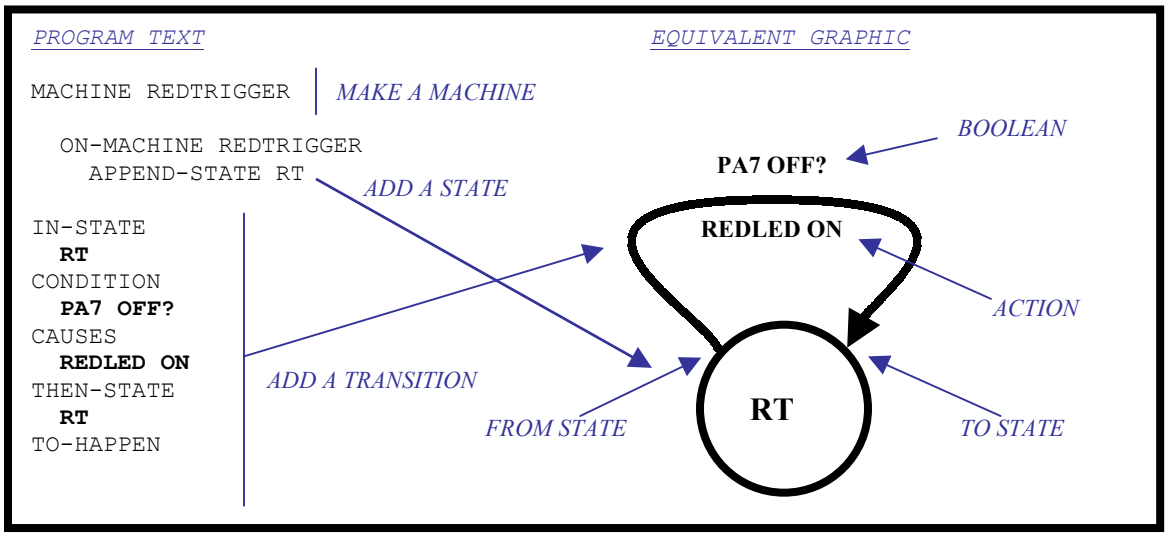
still running, even though all visible changes end at that point. To see that, we'll need to manually reset the LED off so we can see something happen again. Enter.

```
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™ able to type commands like `REDLED OFF` in manually, 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.



In this example, the first program line, we 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:

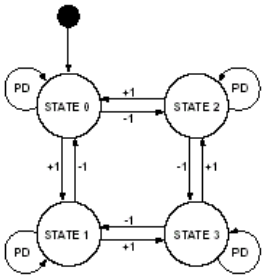
```
IN-STATE RT CONDITION PA7 OFF? CAUSES REDLED ON THEN-STATE RT TO-HAPPEN
```

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 bordering 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. The figure to the right is a hardware Quadrature design with four states.

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. For example: http://www.philipslogic.com/products/hc/pdf/74hc08.pdf

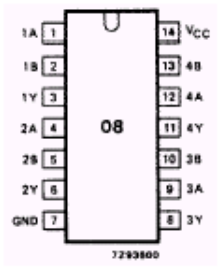


Fig.1 Pin configuration.

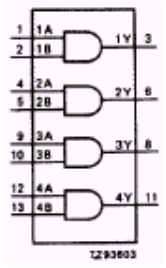


Fig.2 Logic symbol.

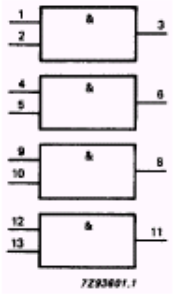


Fig.3 IEC logic symbol.

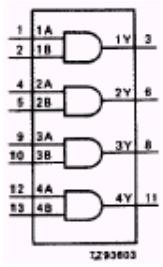


Fig.4 Functional diagram.

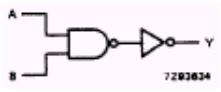


Fig.5 HC logic diagram (one gate).

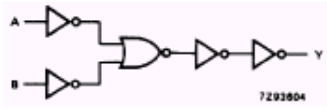


Fig.6 HCT logic diagram (one gate).

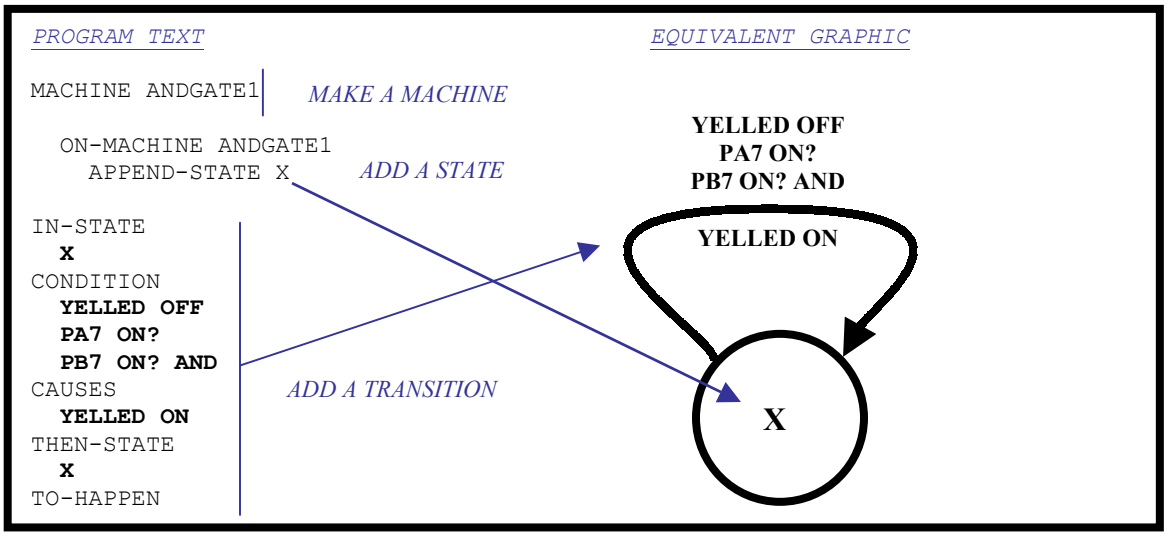
FUNCTION TABLE

INPUTS		OUTPUT
nA	nB	nY
L	L	L
L	H	L
H	L	L
H	H	H

Note

1. H = HIGH voltage level
   L = LOW voltage level

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”.



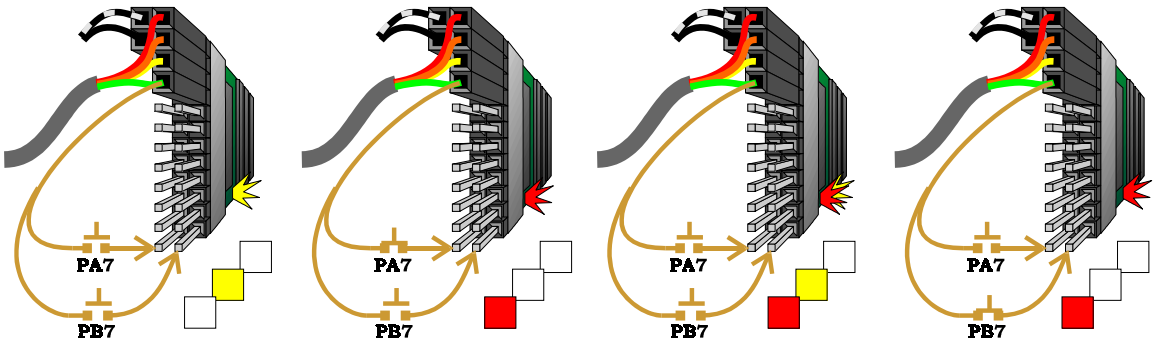
(Now it is worth mentioning, the example is a bit contrived. When you try to make a state machine too simple, you wind up stretching things you shouldn't. This example could have acted exactly like an AND gate if two transitions were used, rather than just one. Instead, a "trick" was used to turn the 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. Let's add some more complexity with another machine sharing the PA7 input.

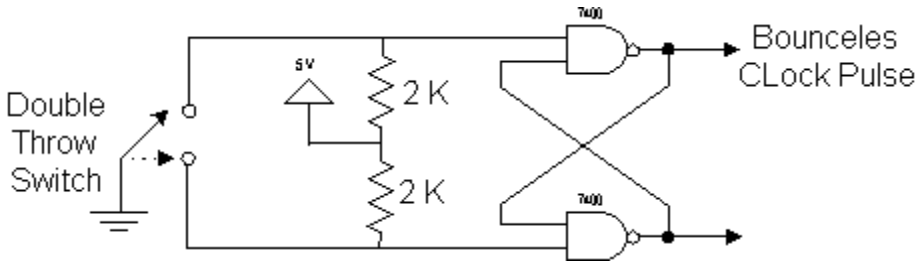
BOUNCELESS

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
```

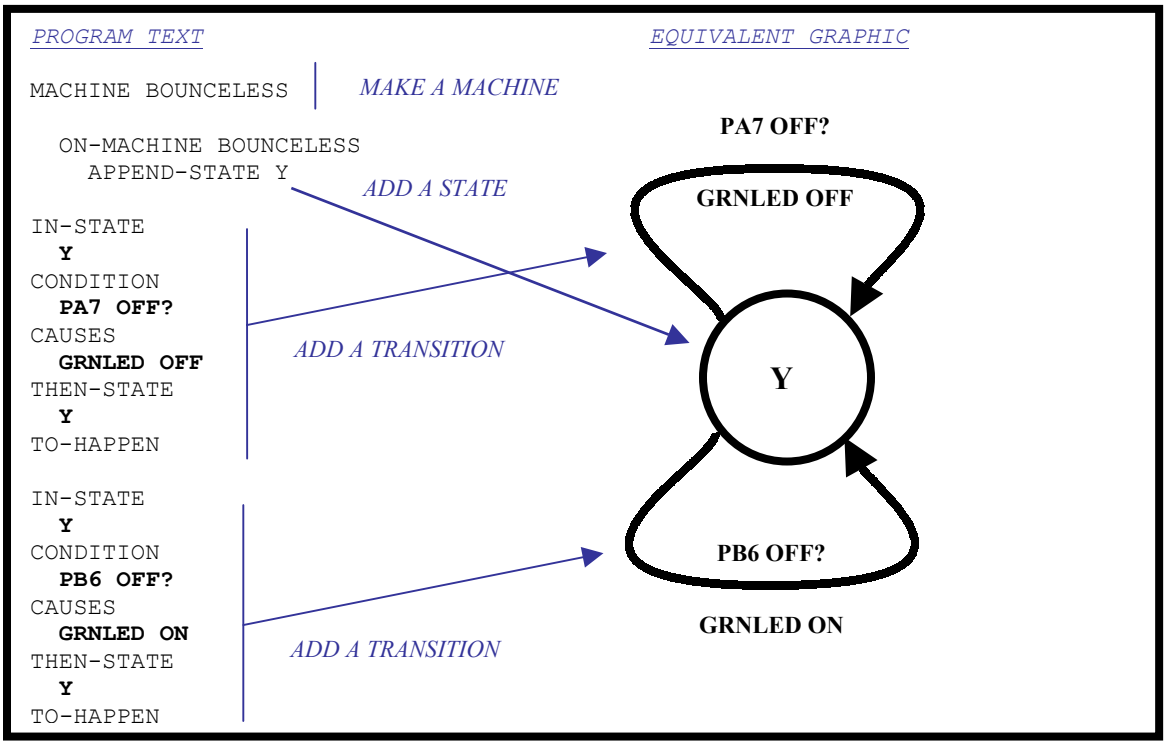
There you have yet another complete design, initialization and installation of a virtual machine in four lines of IsoMax™ code.

Another name for the machine in this program is "a bounceless switch".

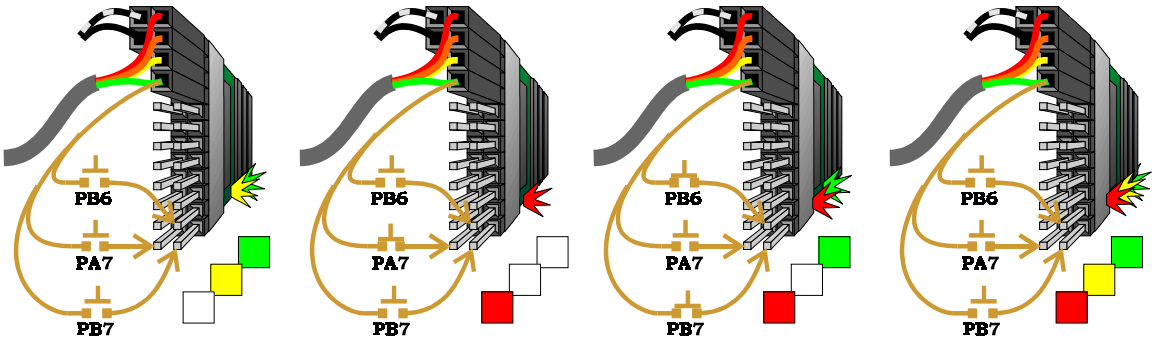


Bounceless switches filter out any noise on their input buttons, and give crisp, one-edge output signals. They do this by toggling state when an input first becomes active, and remaining in that state. If you are familiar with hardware, you might recognize the two gates feed back on each other as a very elementary flip-flop. The flip-flop is a bistable on/off circuit is the basis for a memory cell. The bounceless switch flips when one input is grounded, and will not flip back until the other input is grounded.

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.



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. Further, all three machines share one input, so their behavior is strongly linked.

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.

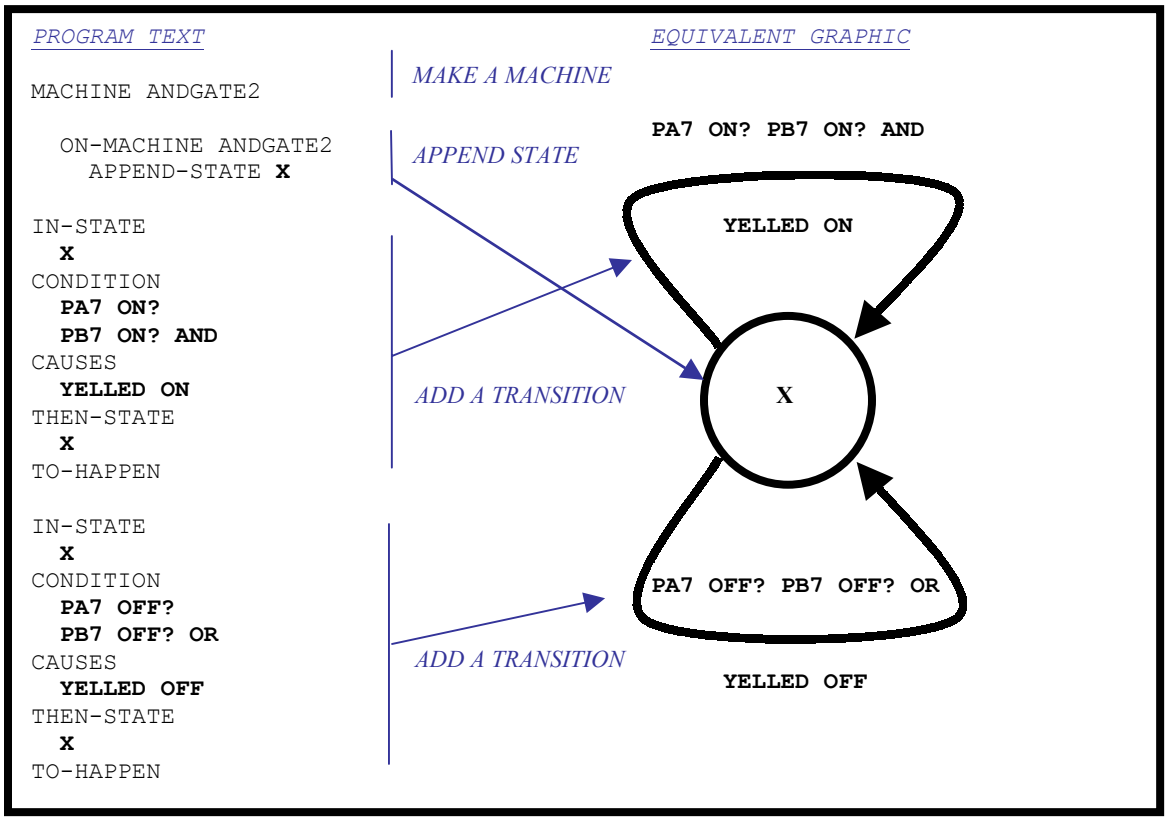
MULTIPLE STATES/MULTIPLE TRANSITIONS

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
```



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" stuff, 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.

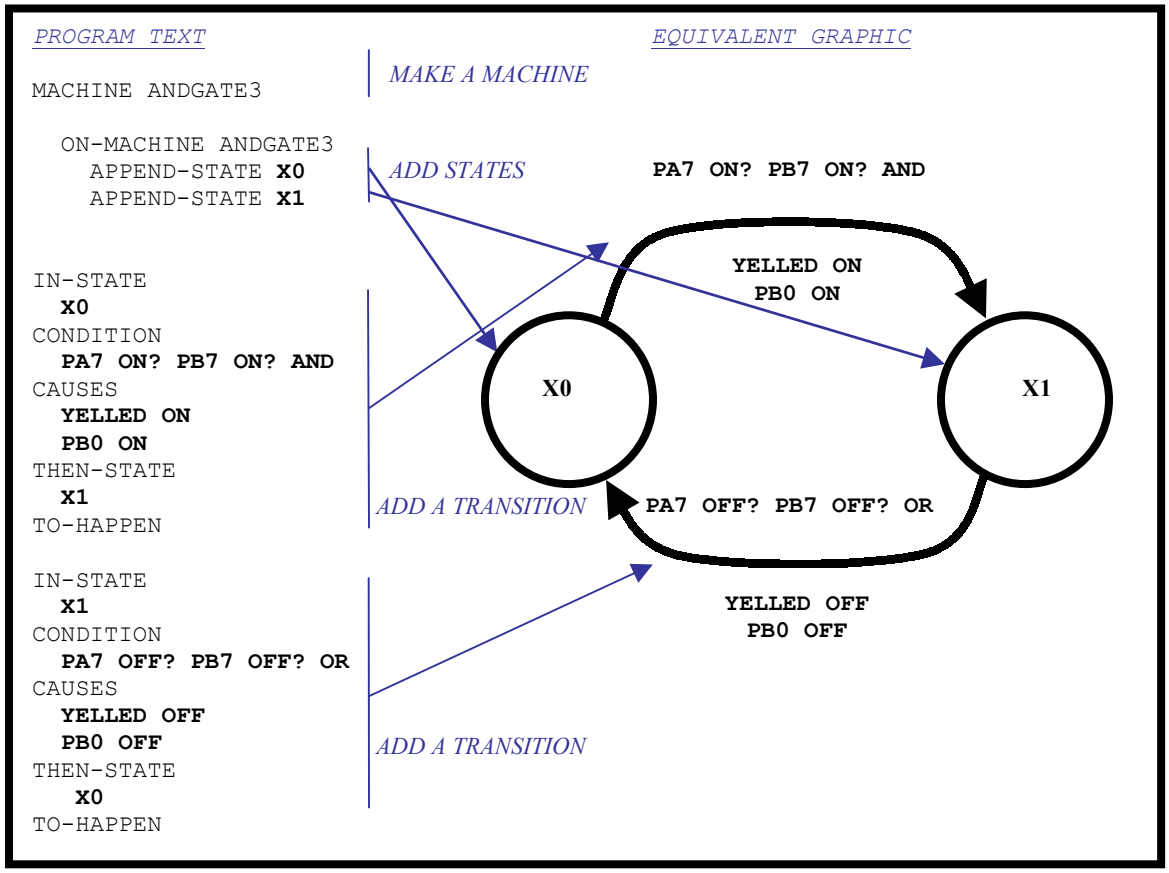
A deeper reason exists that reveals a great truth about state machines. Notice we have used a state machine to simulate a hardware gate. What the AND gate outputs next is completely dependent on what the inputs are next. An AND gate has an output which has no feedback. An AND gate has no memory. State machines can have memory. Their future outputs depend on more than 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
```



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.

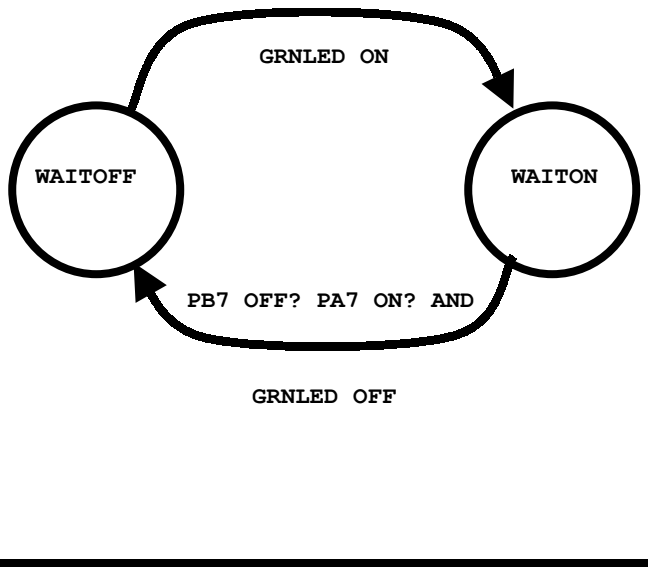
STATE Y	
IN-STATE **Y**	IN-STATE **Y**
CONDITION **PA7 OFF?**	CONDITION **PB6 OFF?**
CAUSES **GRNLED OFF**	CAUSES **GRNLED ON**
THEN-STATE **Y**	THEN-STATE **Y**
TO-HAPPEN	TO-HAPPEN

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.

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



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. 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. (In fact, it is better behaved than the original hardware circuit shown!) 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 **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

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

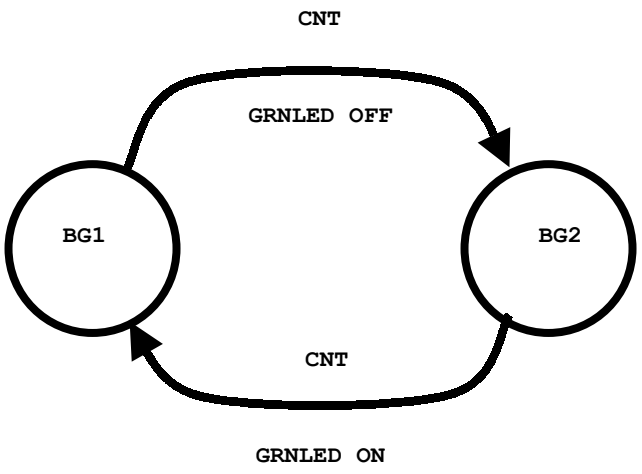
```
: -LOOPVAR <BUILDS HERE P, 1- DUP , , DOES>
  P@ DUP @ 0= IF DUP 1 + @ SWAP ! TRUE ELSE 1-! FALSE THEN ;
100 -LOOPVAR CNT
```

```
IN-STATE
   BG1
CONDITION
   CNT
CAUSES
   GRNLED OFF
THEN-STATE
   BG2
TO-HAPPEN

IN-STATE
   BG2
CONDITION
   CNT
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
CAUSES
  GRNLED OFF
THEN-STATE
  BG2
TO-HAPPEN

IN-STATE
  BG2
CONDITION
  CNT
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

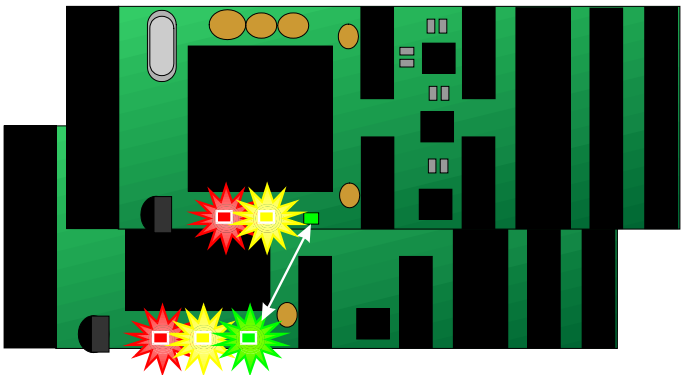
```
IN-STATE BG1 CONDITION CNT CAUSES GRNLED OFF THEN-STATE BG2 TO-HAPPEN
```

```
IN-STATE BG2 CONDITION CNT 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 whichever 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.

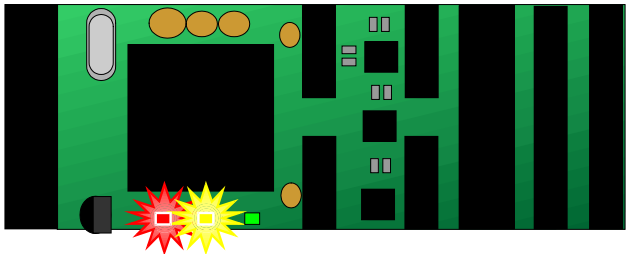
ZIPGRN

```
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
```

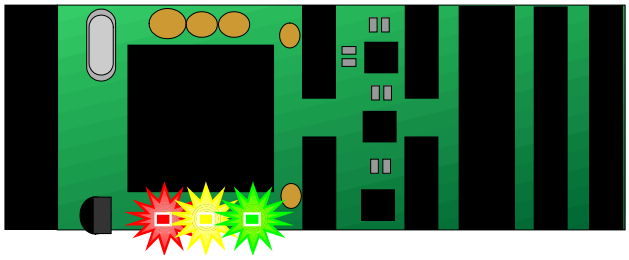
Now rather than install our new machine we're going to test it by running it "by hand" interactively. Type in:

```
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.

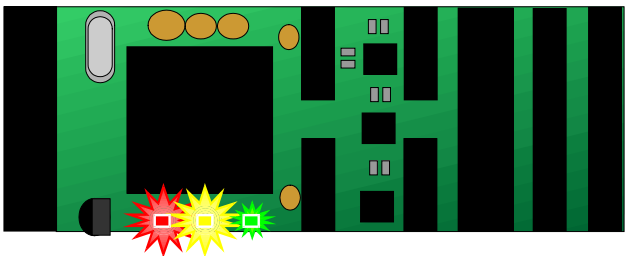
Now with the machine program and tested, we're ready to install the machine into the machine chain. The phrase to install a machine is :

```
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:

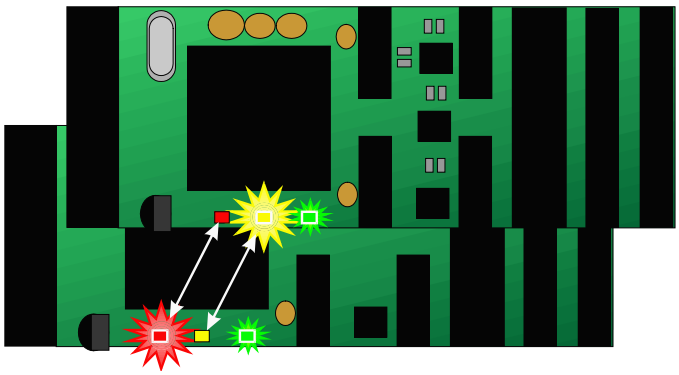
```
MACHINE REDYEL

  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
```

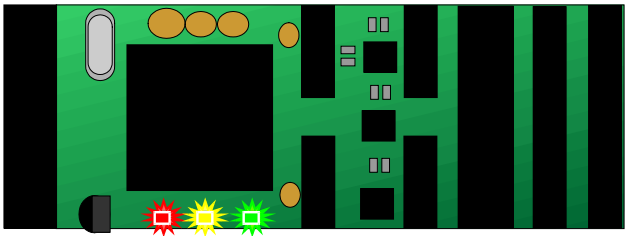
See the red and yellow LED's trade back and forth from on to off and vice versa.



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 proceed 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.

FLASH AND AUTOSTARTING

Here's everything you need to copy an application to Flash and to autostart it. Here, briefly, are the steps:

1. You should start with a clean IsoPod, by doing SCRUB. This will erase the Program Flash and remove any previous autostart patterns.

2. In the program file, each Forth word should be followed by EEWORD. This applies to colon definitions, CODE and CODE-SUB words, constants, variables, "defined" words (those created with <BUILDS..DOES>), and objects (those created with OBJECT).

3. If IMMEDIATE is used, it must come *before* EEWORD (i.e., you must do IMMEDIATE EEWORD and *not* EEWORD IMMEDIATE).

4. For IsoMax code the following rules apply:
  a. MACHINE <name> must be followed by EEWORD.
  b. APPEND-STATE <name> must be followed by EEWORD.
  c. IN-STATE ... TO-HAPPEN (or THIS-TIME or NEXT-TIME) must be followed by IN-EE.
  d. MACHINE-CHAIN ... END-MACHINE-CHAIN must be followed by EEWORD.
  e. ON-MACHINE <name> is *not* followed by any EE command.
[Note that we can make EEWORD and IN-EE automatic, if you want all state machines to be built in Flash and never in RAM.]

5. When the application is complete, you must use SAVE-RAM to preserve the state machine variables in Data Flash. (This does *not* save kernel variables.)

6. Finally you can set the autostart vector in Program Flash. You need to provide an address on a 400h boundary, within unused Program Flash, thus after the end of the application program. (Right now 4700-7DFF is available for applications.) I often use 7C00, near the end of Flash.
Then type
  <address> AUTOSTART <wordname>
E.g., HEX 7C00 AUTOSTART MAIN

The board should now reset into the application program.

PROCEDURAL PROGRAMMING

The FSM portions of IsoMax™ are now covered. What remains to be discussed is the procedural portions of the conditions and actions.

```
END-MACHINE-CHAIN
MACHINE-CHAIN
```

SCHEDULE-RUNS
CYCLES
EVERY
DINT
EINT
STOP-TIMER
TCFOVFLO
TCFTICKS
END-PROC
PROC
AS-TAG
FOR-INPUT
FOR-OUTPUT
WITH-VALUE
SET-MASK
CLR-MASK
XOR-MASK
AND-MASK
DATA-MASK
TEST-MASK
AT-ADDR
IS-STATE?
SET-STATE
TO-HAPPEN
NEXT-TIME
THIS-TIME
THEN-STATE
CAUSES
CONDITION
IN-STATE
ON-MACHINE
APPEND-STATE
MACHINE
CURSTATE

ALLOC
RAM
DEFINE
\
PFMOVE
PFDP
PFERASE
PF!
EEERASE
PTYPE
PCOUNT
P,
PC,
PALLOT
PHERE
PDP
PC!
PC@
P@
P!
TD3

TD2
RS422XCV
RS232XMT
PD0
PD1
PD2
PD3
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
GRNLED
YELLED
REDLED
I/O
OFF
ON
IS
FALSE
TRUE

(
@
C@
!
C!
2@
2!
:
;
+
-
1-!
1+!
+!
*
/
><
SWAP
2OVER
2SWAP
DUP
2DUP
OVER
ROT

2ROT
PICK
ROLL
-ROLL
DROP
2DROP
>R
R>
=
NOT
0=
D0=
0>
0<
U<
<
DU<
D<
D=
>
AND
OR
XOR
IF
THEN
ELSE
BEGIN
UNTIL
REPEAT
WHILE
AGAIN
END
DO
LOOP
+LOOP
K
J
I
R@
LEAVE
EXIT
KEY
EMIT
?TERMINAL
S->D
ABS
DABS
MIN
DMIN
MAX
DMAX
SPACES
DEPTH
CR
TYPE
COUNT
-TRAILING

1+
2+
1-
2-
2/
2*
D+
D-
D2/
/MOD
MOD
*/MOD
*/
UM*
UM/MOD
NEGATE
DNEGATE
CONSTANT
VARIABLE
2CONSTANT
2VARIABLE

SF!
SF@
FTAN
FCOS
FSIN
FATAN2
FATAN
F?
FSQRT
F2/
F2*
F.S
FNUMBER
E.
F.
(E.)
(F.)
F**
FALOG
FEXP
2**X
FLN
FLOG
LOG2
ODD-POLY
POLY
FLOOR
FROUND
FLITERAL
PI
E
PLACES
FLOAT+
FLOATS
FVARIABLE

```
FCONSTANT
F,
F!
F@
FABS
FMIN
FMAX
F<
F0<
F0=
FNEGATE
F>D
S>F
D>F
F/
F*
F-
F+
FDROP
FSWAP
FOVER
FDUP
FNIP
FDEPTH
FSP
FSP0

TOGGLE
SP!
RP@
RP!
UABORT
WARNING
R0
SMUDGE
DLITERAL
MESSAGE
ERROR
?ERROR
?COMP
?EXEC
?PAIRS
?CSP
?STACK
@!
@@
EXECUTE
SP@
CMOVE>
CMOVE
;S
CODE-SUB
CODE
END-CODE
USER
.
.R
```

D.
U.
U.R
D.R
#S
#
SIGN
#>
<#
?
EXPECT
QUERY
BL
STATE
CURRENT
CONTEXT
BLK
DP
FLD
DPL
>IN
BASE
S0
TIB
#TIB
SPAN
C/L
PAD
HERE
ALLOT
,

C,
SPACE
?DUP
TRAVERSE
LATEST
COMPILE
[
]
HEX
DECIMAL
;CODE
<BUILDS
DOES>
."
.(
FILL
ERASE
BLANK
HOLD
WORD
CONVERT
NUMBER
FIND
ID.
CREATE

[COMPILE]
LITERAL
INTERPRET
IMMEDIATE
RECURSE
>MARK
<MARK
>RESOLVE
<RESOLVE
:CASE
'
[']
LFA
>BODY
CFA
NFA
PFAPTR
B/BUF
AUTOSTART
UNDO
FORGET
DUMP
.S
WORDS
QUIT
ABORT"
ABORT
COLD
BRANCH
?BRANCH
AT04
EEWORD
EEMOVE
EEC!
EE!
EDP
EDELAY
FLASH
EXRAM
Seed
FORTH-83

SOFTWARE

IsoMax™ is an interactive, real time control, computer language based on the concept of the State Machine.

WORD SYNTAX

STATE-MACHINE <name-of-machine>

ON-MACHINE <name-of-machine>
    APPEND-STATE <name-of-new-state>
    ...
    APPEND-STATE <name-of-new-state> WITH-VALUE <n> AT-ADDRESS <a> AS-TAG

IN-STATE <parent-state-name> CONDITION ...boolean computation... CAUSES <compound action> THEN-STATE <next-state-name> TO-HAPPEN

DEFINE <word-name> TEST-MASK <n> DATA-MASK <n> AT-ADDRESS <a> FOR-INPUT

DEFINE <word-name> SET-MASK <n> CLR-MASK <n> AT-ADDRESS <a> FOR-OUTPUT

DEFINE <word-name> PROC ...forth code... END-PROC

DEFINE <word-name> COUNTDOWN-TIMER
<n> TIMER-INIT <timer-name>

EVERY <n> CYCLES SCHEDULE-RUNS ALL-TASKS

Under construction…

WITH-VALUE (-- 7100) stacks the tag 7100.
AT-ADDRESS (-- 7001) stacks the tag 7001. This will be topmost after ORDER.
AS-TAG (tag n tag n --)
    Requires tags 7100,7001. Requires the latest word to be a State word. If it is, removes DUMMYTAG, 0 and replaces them with Address, Value.

THIS-TIME (spfa --) *previously TO-HAPPEN ?*

Requires CSP=HERE. Requires the given word to be a State word. Then:
Removes last compiled cell. Compiles the CFA of the given State word. Compiles PTHIST.

NEXT-TIME (spfa --)
Requires CSP=HERE. Requires the given word to be a State word. Then:
Removes last compiled cell. Compiles the CFA of the given State word. Compiles PNEXTT.

SET-STATE (spfa --)
Given the pfa of a State word on the stack. Requires the given word to be a State word. Then:
Fetches the thread pointer and RAM pointer from the State word, and stores the thread pointer in the RAM pointer.

IS-STATE? (spfa --)
Given the pfa of a State word on the stack. Requires the given word to be a State word. Then:
Fetches the thread pointer and RAM pointer from the State word. Returns true if the current state of the machine is this state.

IN-EE

TIMING CONTROL

EVERY (-- 6000) stacks the value 6000.
CYCLES (-- 9000) stacks the value 9000.

SCHEDULE-RUNS *not defined in source file*
ALL-TASKS *not defined in source file*
COUNTDOWN-TIMER *not defined in source file*
TIMER-INIT *not defined in source file*

INPUT/OUTPUT TRINARIES

DEFINE <word-name> (-- 1111)
Creates a new word in the Forth dictionary (CREATE SMUDGE) and stacks the pair-tag 1111.

PROC *not defined in source file*
END-PROC *not defined in source file*

TEST-MASK (-- 7002) stacks the tag 7002.
DATA-MASK (-- 7004) stacks the tag 7004.

FOR-INPUT (1111 tag n tag n tag n --)
If tags 7001, 7002, 7004 are stacked, compiles Address, Test-Mask (byte), and Data-Mask (byte), then changes the code field of the latest word to XCPAT. Requires pair-tag 1111.

XCPAT
Fetches the data byte from the stored Address, masks it with the Test-Mask, and xors it with the Data-Mask. If the result is zero (equal), stacks TRUE, else stacks FALSE.

AND-MASK (-- 7008) stacks the tag 7008.

XOR-MASK (-- 7010) stacks the tag 7010.

CLR-MASK (-- 7020) stacks the tag 7020.
SET-MASK (-- 7040) stacks the tag 7040.

FOR-OUTPUT (1111 tag n tag n tag n --)

If tags 7001, 7008, 7010 are stacked, compiles Address, And-Mask (byte), and Xor-Mask (byte), then changes the code field of the latest word to AXOUT.
If tags 7001, 7020, 7040 are stacked, compiles Address, Clr-Mask (byte), and Set-Mask (byte), then changes the code field of the latest word to SROUT.
Requires pair-tag 1111.

REGISTERS

Under construction…

(BASE REGISTERS)
0C00 SIM
0C40 PFIU2
0D00 TMRA
0D20 TMRB
0D40 TMRC
0D60 TMRD
0D80 CAN
0E00 PWMA
0E20 PWMB
0E40 DEC0
0E50 DEC1
0E60 ITCN
0E80 ADCA
0EC0 ADCB
0F00 SCI0
0F10 SCI1
0F20 SPI
0F30 COP
0F40 PFIU
0F60 DFIU
0F80 BFIU
0FA0 CLKGEN
0FB0 GPIOA
0FC0 GPIOB
0FE0 GPIOD
0FF0 GPIOE

(TIMER REGISTERS. OFFSET IS CHANNEL * 8)

0 CMP1
1 CMP2
2 CAP
3 LOAD
4 HOLD
5 CNTR
6 CTRL
7 SCR

(GPIO)

0 PUR
1 DR
2 DDR
3 PER
4 IAR
5 IENR
6 IPOLR
7 IPR
8 IESR

(A/D CONVERTER)

0 ADCR1
1 ADCR2
2 ADZCC
3 ADLST1
4 ADLST2
5 ADSDIS
6 ADSTAT
7 ADLSTAT
8 ADZCSTAT
9 ADRSLT0
A ADRSLT1
B ADRSLT2
C ADRSLT3
D ADRSLT4
E ADRSLT5
F ADRSLT6
10 ADRSLT7
11 ADLLMT0
12 ADLLMT1
13 ADLLMT2
14 ADLLMT3
15 ADLLMT4
16 ADLLMT5
17 ADLLMT6
18 ADLLMT7
19 ADHLMT0
1A ADHLMT1
1B ADHLMT2
1C ADHLMT3
1D ADHLMT4
1E ADHLMT5
1F ADHLMT6
20 ADHLMT7

21 ADOFS0
22 ADOFS1
23 ADOFS2
24 ADOFS3
25 ADOFS4
26 ADOFS5
27 ADOFS6
28 ADOFS7

(PWM)

0 PMCTL
1 PMFCTL
2 PMFSA
3 PMOUT
4 PMCNT
5 PWMCM
6 PWMVAL0
7 PWMVAL1
8 PWMVAL2
9 PWMVAL3
A PWMVAL4
B PWMVAL5
C PMDEADTM
D PMDISMAP1
E PMDISMAP2
F PMCFG
10 PMCCR
11 PMPORT

(QUAD)

0 DECCR
1 FIR
2 WTR
3 POSD
4 POSDH
5 REV
6 REVH
7 UPOS
8 LPOS
9 UPOSH
A LPOSH
B UIR
C LIR
D IMR

E TSTREG

(SCI)

0 SCIBR
1 SCICR
2 SCISR
3 SCIDR

(SPI)

0 SPSCR
1 SPDSR
2 SPDRR
3 SPDTR

IsoPod™ MEMORY MAP

DATA MEMORY

0000 04E6	Data RAM (Kernel)
04E7 07FF	Data RAM (User)
0800 0BFF	reserved
0C00 0FFF	peripherals
1000 1BFF	Data Flash (Kernel)
1C00 1FFF	Data Flash (User)

PROGRAM MEMORY

0000 31FF	Program Flash (Kernel)
3200 7DFF	Program Flash (User)
7E00 7FDF	Program RAM (User)
7FE0 7FFF	Program RAM (Kernel*)

* Program RAM is used by the kernel only for the Flash programming routines. This space is otherwise available for the user.

HARVARD MEMORY MODEL

The IsoPod Processor uses a "Harvard" memory model, which means that it has separate memories for Program and Data storage. Each of these memory spaces uses a 16-bit address, so there can be 64K 16-bit words of Program ("P") memory, and 64K 16-bit words of Data ("X") memory.

MEMORY OPERATORS

Most applications need to manipulate data, so the memory operators use Data space. These include

```
@   !   C@   C!   +!   HERE   ALLOT   ,   C,
```

Occasionally you will need to manipulate Program memory. This is accomplished through a separate set of memory operators having a "P" prefix:

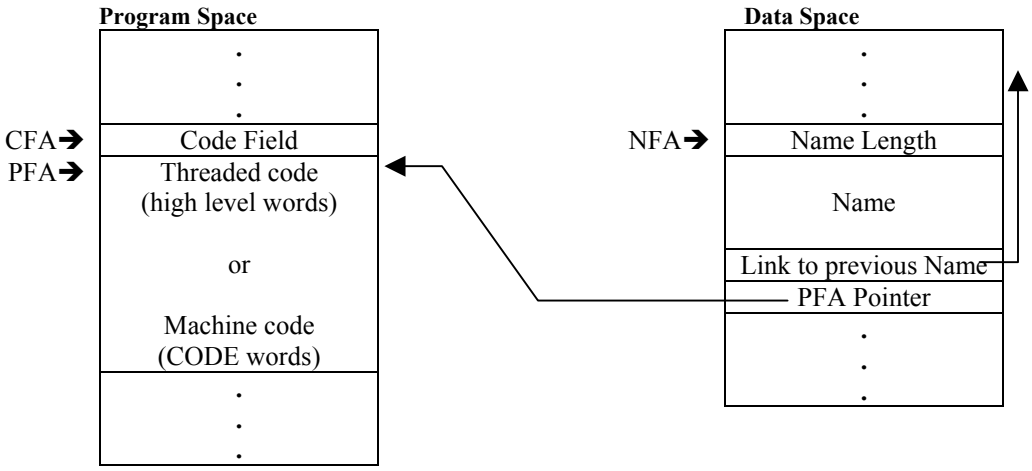
```
P@   P!   PC@   PC!   PHERE   PALLOT   P,   PC,
```

Note that on the IsoPod™, the smallest addressable unit of memory is one 16-bit word. This is the unpacked character size. This is also the "cell" size used for arithmetic and addressing. Therefore, `@` and `C@` are equivalent, and `!` and `C!` are equivalent.

WORD STRUCTURE

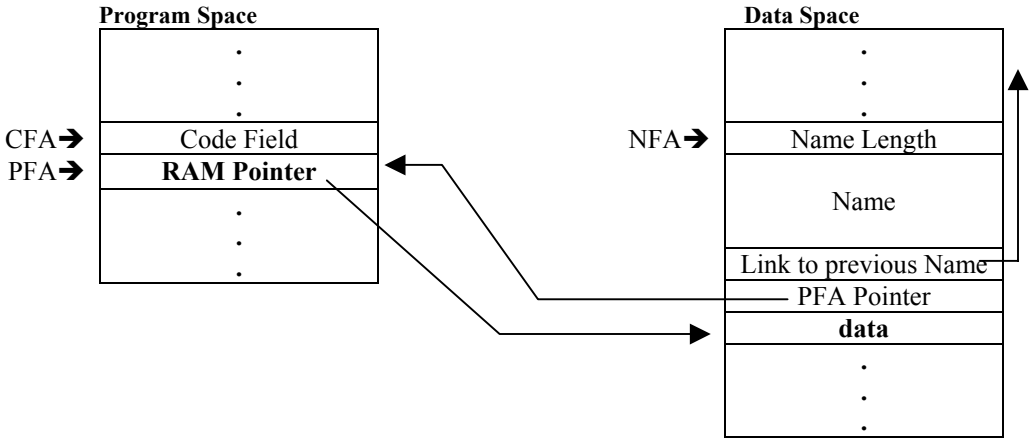
The executable "body" of a IsoMax™ word is kept in Program space. This includes the Code Field of the word, and the threaded definition of high-level words or the machine code definition of CODE words.

The "header" of a IsoMax™ word is kept in Data space. This includes the Name Field, the Link Field, and the PFA Pointer.



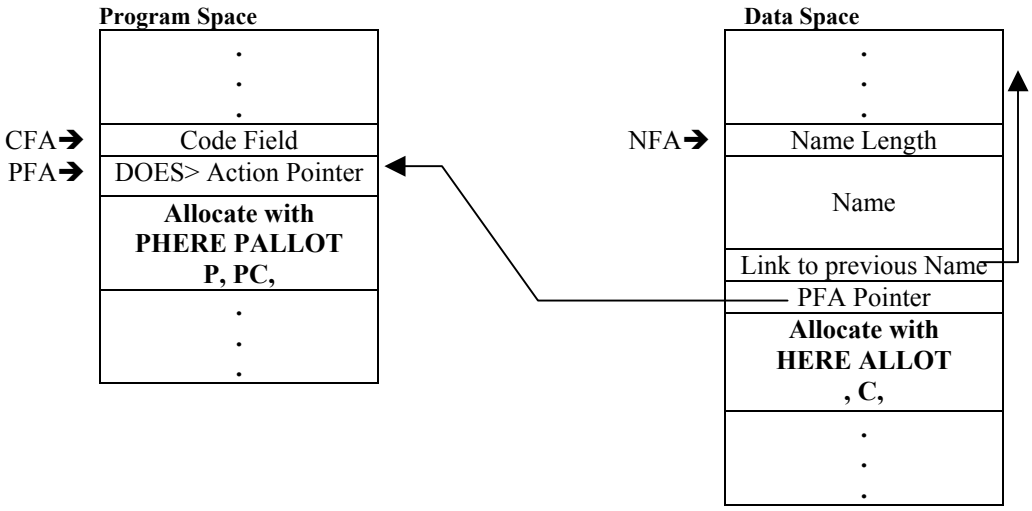
VARIABLES

Since the Program space is normally ROM, and variables must reside in RAM and in Data space, the "body" of a VARIABLE definition does not contain the data. Instead, it holds a pointer to a RAM location where the data is stored.



<BUILDS DOES>

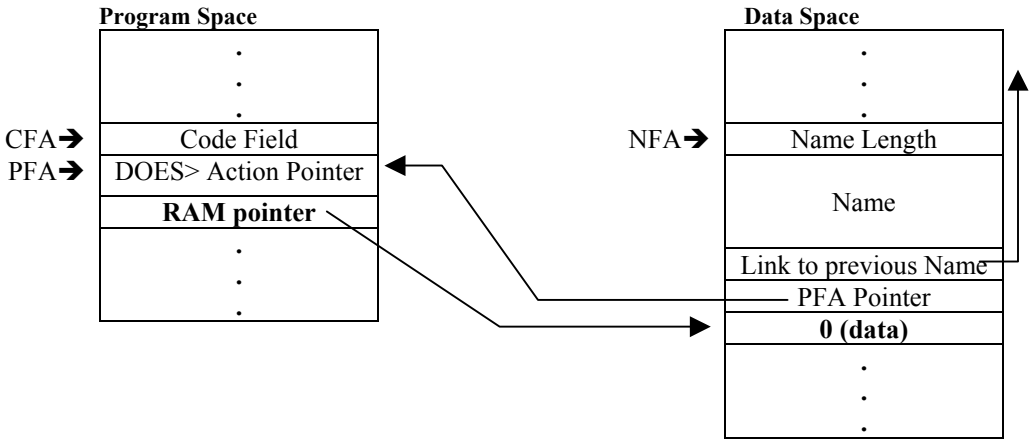
"Defining words" created with <BUILDS and DOES> may have a variety of purposes. Sometimes they are used to build Data objects in RAM, and sometimes they are used to build objects in ROM (i.e., in Program space). In the <BUILDS code you can allocate either space by using the appropriate memory operators.



For maximum flexibility, DOES> will leave on the stack the address *in Program space* of the user-allocated data. If you need to allocate data in Data space, you must also store (in Program space) a pointer to that data. For example, here is how you might define VARIABLE using <BUILDS and DOES>.

```
: VARIABLE
  <BUILDS     Defines a new Forth word, header and empty body;
    HERE      gets the address in Data space (HERE) and appends that to Program space;
    0  ,      appends a zero cell to Data space.
  DOES>       The "run-time" action will start with the Program address on the stack;
    P@        fetch the cell stored at that address (a pointer to Data) and return that.
;
```

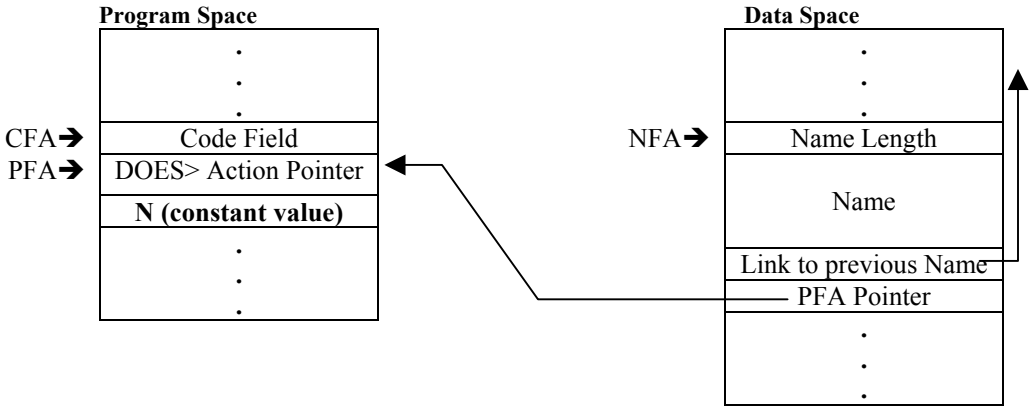
This constructs the following:



Words with constant data, on the other hand, can be allocated entirely in Program space. Here's how you might define CONSTANT:

```
: CONSTANT  ( n -- )
  <BUILDS     Defines a new Forth word, header and empty body;
    P,                appends the constant value (n) to Program space.
  DOES>       The "run-time" action will start with the Program address on the stack;
    P@        fetch the cell stored at that address (the constant) and return that.
;
```

This constructs the following:



IsoPod™ Reset Sequence

The IsoPod employs a flexible initialization that gives you many options for starting and running application programs. Sophisticated applications can elect to run with or without IsoMax, and with the default or custom processor initialization. This requires some knowledge of the steps that the IsoPod takes upon a processor reset:

1. Perform basic CPU initialization. This includes the PLL clock generator and the RS232 serial port.

2. Do the `QUICK-START` routine. If a `QUICK-START` vector is present in RAM, execute the corresponding routine. `QUICK-START` is designed to be used before any other startup code, normally just to provide some additional initialization. In particular, this is performed before RAM is re-initialized. This gives you the opportunity to save any RAM status, for example on the occurrence of a watchdog reset. Note that a power failure which clears the RAM will also clear the `QUICK-START` vector.

3. Stop IsoMax. This is in case of a "software reset" that would otherwise leave the timer running.

4. Check for "autostart bypass." Configure the SCLK/PE4 pin as an input with pullup resistor. If the SCLK/PE4 pin then reads a continuous "0" (ground level) for 1 millisecond, skip the autostart sequence and "coldstart" the IsoPod. This will initialize RAM to factory defaults and start the IsoMax interpreter.

This is intended to recover from a situation where an autostart application locks up the IsoPod. Simply jumper the SCLK/PE4 pin to ground, and reset the IsoPod. This will reset the RAM and start the interpreter, but please note that it will *not* erase any Flash ROM. Flash ROM can be erased with the `SCRUB` command from the IsoMax interpreter.

This behavior should be kept in mind when designing hardware around the IsoPod. If the IsoPod is installed as an SPI master, or if the SCLK/PE4 pin is used as a programmed output, there will be no problem. If the IsoPod is installed as an SPI slave, the presence of SPI clock pulses will not cause a coldstart, but a coldstart *will* happen if SCLK is held low in the "idle" state and a CPU reset occurs. For this reason, if the IsoPod is an SPI slave, we recommend configuring the SPI devices with CPOL=1, so the "idle" state of SCLK is high. If the SCLK/PE4 pin is used as a programmed input, avoid applications where this pin might be held low when a CPU reset occurs.

If SCLK/PE4 is *not* grounded, proceed with the autostart sequence.

5. Check the contents of RAM and initialize as required.

a. If the RAM contents are valid[1], use them. This will normally be the case if the CPU is reset with no power cycle, e.g., reset by MaxTerm, a watchdog, or an external reset signal.

b. If the RAM contents are invalid, load the `SAVE-RAM` image from Data Flash ROM. If this RAM image is valid, use it. This gives you a convenient method to initialize your application RAM.

c. If the Flash ROM contents are invalid, then reinitialize RAM to factory defaults. Note that this will reset the dictionary pointer but will *not* erase any Flash ROM.

6. Look for a "boot first" routine. Search for an $A44A pattern in Program Flash ROM. The search looks at 1K ($400) boundaries, starting at Program address $400 and proceeding to $7C00. If found, execute the corresponding "boot first" routine. IsoMax is *not* running at this point.

a. If the "boot first" routine never exits, only it will be run.

b. If the "boot first" routine exits, or if no $A44A pattern is found, continue the autostart sequence.

7. Start IsoMax with an "empty" list of state machines. After this, you can begin `INSTALL`ing state machines. Any state machines `INSTALL`ed before this point will be disabled.

8. Look for an "autostart" routine. Search for an $A55A pattern in Program Flash ROM. The search looks at 1K ($400) boundaries, starting at Program address $400 and proceeding to $7C00. If found, execute the corresponding "autostart" routine.

a. If the "autostart" routine never exits, only it will be run. (Of course, any IsoMax state machines `INSTALL`ed by this routine will also run.)

b. If the "autostart" routine exits, or if no $A55A pattern is found, start the IsoMax interpreter.

---

[1] RAM is considered "valid" if the program dictionary pointer is within the Program Flash ROM address space, the version number stored in RAM matches the kernel version number, and the SYSTEM-INITIALIZED variable contains the value $1234.

In summary:

Use the `QUICK-START` vector if you need to examine uninitialized RAM, or for chip initialization which must occur immediately.
Use an $A44A "boot first" vector for initialization which must *precede* IsoMax activation, but which needs initialized RAM.
Use an $A55A "autostart" vector to install IsoMax state machines, and for your main application program.
To bypass the autostart sequence, jumper SCLK/PE4 to ground.

Object Oriented Extensions

These words provide a fast and compact object-oriented capability to MaxForth. It defines Forth words as "methods" which are associated only with objects of a specific class.

Action of an Object

An object is very much like a <BUILDS DOES> defined word. It has a user-defined data structure which may involve both Program ROM and Data RAM. When it is executed, it makes the address of that structure available (though not on the stack...more on this in a moment).

What makes an object different is that there is a "hidden" list of Forth words which can only be used by that object (and by other objects of the same class). These are the "methods," and they are stored in a private wordlist. *Note that this is not the same as a Forth "vocabulary." Vocabularies are not used, and the programmer never has to worry about word lists.*

Each method will typically make several references to an object, and may call other methods for that object. If the object's address were kept on the stack, this would place a large burden of stack management on the programmer. To make object programming simpler *and* faster, the address of the current object is stored in a variable, `OBJREF`. The contents of this variable (the address of the current object) can always be obtained with the word `SELF`.

When *executed (interpreted)*, an object does the following:
1. Make the "hidden" word list of the object available for searching.
2. Store the object's address into OBJREF.

After this, the private methods of the object can be executed. (These will remain available until an object of a different class is executed.)

When *compiled*, an object does the following:
1. Make the "hidden" word list of the object available for searching.
2. Compile code into the current definition which will store the object's address into OBJREF.

After this, the private methods of the object can be compiled. (These will remain available until an object of a different class is compiled.) *Note that both the object address and the method are resolved at compile time. This is "early binding" and results in code that is as fast as normal Forth code.*

In either case, the syntax is identical:

```
object method
```

For example:

```
REDLED TOGGLE
```

Defining a new class

BEGIN-CLASS name

Words defined here will only be visible to objects of this class.
These will normally be the "methods" which act upon objects of this class.

PUBLIC

Words defined here will be visible at all times.
These will normally be the "objects" which are used in the main program.

END-CLASS name

Defining an object

OBJECT name This defines a Forth word "name" which will be an object of the current class. The object will initially be "empty", that is, it will have no ROM or RAM allocated to it. The programmer can add data structure to the object using `P, , PALLOT` and `ALLOT`, in the same manner as for <BUILDS DOES> words. *Like <BUILDS DOES>, the action of an object is to leave its **Program** memory address.*

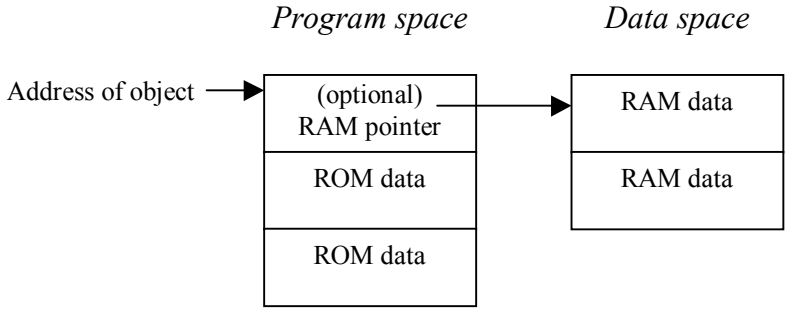
Referencing an object

SELF This will return the address of the object last executed. *Note that this is an address in **Program** memory. If the object will use Data RAM, it is the responsibility of the programmer to store a pointer to that RAM space. See the example below.*

Object Structure

An object may have associated data in both Program and Data spaces. This allows ROM parameters which specify the object (e.g., port numbers for an I/O object); and private variables ("instance variables") which are associated with the object. By default, objects return their Program (ROM) address. If there are RAM variables associated with the object, a pointer to those variables must be included in the ROM data.

Object data structure



Note that also `OBJECT` creates a pointer to Program space, it does not reserve *any* Program or Data memory. That is the responsibility of the programmer. This is done in the same manner as the <BUILDS clause of a <BUILDS DOES> definition, using `P,` or `PALLOT` to add cells to Program space and `,` or `ALLOT` to add cells to Data space. The programmer can use `OBJECT` to build a custom defining word for each class. See the example below.

Example using ROM and RAM

This is an example of an object which has both ROM data (a port address) and RAM data (a timebase value).

```
BEGIN-CLASS TIMERS
  : TIMER ( a -- )  OBJECT  HERE 1 ALLOT P,  P, ;
PUBLIC
  0D00 TIMER TA0
  0D08 TIMER TA1
END-CLASS TIMERS
```

The word `TIMER` expects a port address on the stack. It builds a new (empty) `OBJECT`. Then it reserves one cell of Data RAM (`1 ALLOT`) and stores the starting address of that RAM (`HERE`) into Program memory (`P,`). This builds the RAM pointer as shown above. Finally, it stores the I/O port address "a" into the second cell of Program memory (the second `P,`). *Each* object built with `TIMER` will have its own copy of this data structure.

After the object is executed, `SELF` will return the address of the Program data for that object. Because we've stored a RAM pointer as the first Program cell, the phrase `SELF P@` will return the address of the RAM data for the object. *It is not required that the first Program cell be the RAM pointer, but this is strongly recommended as a programming convention for all objects using RAM storage.*

Likewise, `SELF CELL+ P@` will return the I/O port address associated with this object (since that was stored in the second cell of Program memory by `TIMER`).

We can simplify programming by making these phrases into Forth words. We can also build them into other Forth words. All of this will normally go in the "private" class dictionary:

```
BEGIN-CLASS TIMERS
  : TIMER      ( a -- )  OBJECT  HERE 1 ALLOT P,  P, ;

  : TMR_PERIOD ( -- a )  SELF P@ ;    ( RAM variable for
this timer)
  : BASEADDR   ( -- a )  SELF CELL+ P@ ;  ( I/O addr for
this timer)
  : TMR_SCR    ( -- a )  BASEADDR 7 + ;   ( Control
register )

  : SET-PERIOD ( n -- )  TMR_PERIOD ! ;
  : ACTIVE-HIGH ( -- )   0202 TMR_SCR CLEAR-BITS ;
PUBLIC
  0D00 TIMER TA0      ( Timer with I/O address 0D00 )
  0D08 TIMER TA1      ( Timer with I/O address 0D08 )
END-CLASS TIMERS
```

After this, the phrase `100 TA0 SET-PERIOD` will store the RAM variable for timer object TA0, and `200 TA1 SET-PERIOD` will store the RAM variable for timer object TA1. `TA0 ACTIVE-HIGH` will clear bits in timer A0 (at port address 0D07), and `TA1 ACTIVE-HIGH` will clear bits in timer A1 (at port address 0D0F).

In a `WORDS` listing, only `TA0` and `TA1` will be visible. But after executing `TA0` or `TA1`, all of the words in the `TIMERS` class will be found in a dictionary search.

Because the "methods" are stored in private word lists, you can re-use method names in different classes. For example, it is possible to have an `ON` method for timers, a different `ON` method for GPIO pins, a third `ON` method for PWM pins, and so on. When the object is named, it will automatically select the correct set of methods to be used! Also, if a particular method has *not* been defined for a given object, you will get an error message if you attempt to use that method with that object. (One caution: if there is word in the Forth dictionary with the same name, and there is no method of that name, the Forth word will be found instead. An example of this is `TOGGLE`. If you have a `TOGGLE` method, that will be compiled. But if you use an object that doesn't have a `TOGGLE` method, Forth's `TOGGLE` will be compiled. *For this reason, methods should **not** use the same names as "ordinary" Forth words.*)

Because the "objects" are in the main Forth dictionary, they must all have unique names. For example, you can't have a Timer named `A0` and a GPIO pin named `A0`. You must give them unique names like `TA0` and `PA0`.

GPIO Bit I/O Class

These words support the GPIO I/O of the DSP56F80x. The following GPIO pins are defined as objects:

```
PA7   PA6   PA5   PA4   PA3   PA2   PA1   PA0
PB7   PB6   PB5   PB4   PB3   PB2   PB1   PB0
PD3   PD2   PD1   PD0
REDLED  YELLED  GRNLED
```

For each pin, the following methods can be performed:

`ON` Makes the pin an output, and outputs a '1' (high level).
`OFF` Makes the pin an output, and outputs a '0' (low level).
`TOGGLE` Makes the pin an output, and inverts its level.
`n SET` Stores a T/F value to the pin, e.g., `1 PA0 SET`. Any nonzero value is "true."
`GETBIT` Makes the pin an input, and returns pin value (as a bit mask).
`ON?` Makes the pin an input, and returns true if pin is '1' (high level).
`OFF?` Makes the pin an input, and returns true if pin is '0' (low level).
`IS-INPUT` Makes pin an input (hi-Z).
`IS-OUTPUT` Makes pin an output. Pin will output the last programmed level.

Examples of use:

```
PA0 OFF   ( output a low level on PA0 )
0 PA0 SET ( also outputs a low level on PA0 )
REDLED ON ( output a high level, turn the red LED on )
PD3 ON?   ( check if PD3 is a logic '1' )
```

GPIO Byte I/O Class

These words support the GPIO I/O of the DSP56F80x as bytes. The following GPIO ports are defined as objects:

```
PORTA   PORTB
```

For each pin, the following methods can be performed:

`IS-INPUT` Makes port an input (hi-Z).
`IS-OUTPUT` Makes port an output. Pin will output the last programmed level.
`PUTBYTE` Makes port an output, and outputs the given byte (8 bits).
`GETBYTE` Makes port an input, and reads it as a byte (8 bits).

Examples of use:

```
55 PORTA PUTBYTE    ( output 55 to GPIO Port A )
PORTB GETBYTE .     ( read GPIO Port B and type its numeric
value )
```

Timer I/O Class

These words support the Counter/Timers of the DSP56F80x. The following timers are defined as objects:

```
TA0   TA1   TA2   TA3
TB0   TB1   TB2   TB3
TC0   TC1   TC2   TC3
TD0   TD1   TD2
```

For each Counter/Timer, the following methods can be performed:

`ON` Makes the counter/timer pin an output, and outputs a '1' (high level).
`OFF` Makes the counter/timer pin an output, and outputs a '0' (low level).
`TOGGLE` Makes the counter/timer pin an output, and inverts its level.
`n SET` Stores a T/F value to the pin, e.g., `1 TA0 SET`. Any nonzero value is "true."
`GETBIT` Makes the counter/timer pin an input, and returns pin value (as a bit mask).
`ON?` Makes the counter/timer pin an input, and returns true if pin is '1' (high level).
`OFF?` Makes the counter/timer pin an input, and returns true if pin is '0' (low level).

The following methods can be used to generate PWM signals and to measure pulse width:

`ACTIVE-HIGH` Makes the pin "active high" for PWM output or input. For output, `PWM-OUT` will control the *high* pulse width. For input, `PWM-IN` will measure the width of the *high* pulse. The reset default is `ACTIVE-HIGH`.
`ACTIVE-LOW` Makes the pin "active low" for PWM output or input. For output, `PWM-OUT` will control the *low* pulse width. For input, `PWM-IN` will measure the width of the *low* pulse.
`n PWM-PERIOD` Specifies the period (frequency) of the PWM output. Values from 100 to FFFF hex are valid. The counter frequency is 2.5 MHz; FFFF hex corresponds to a period of 26.214 msec (38 Hz). `PWM-PERIOD` must be specified before using `PWM-OUT`.
`n PWM-OUT` Makes the counter/timer pin an output, and outputs a continuous PWM signal with the given duty cycle. Values from 0 to FFFF hex are valid. 0 is a duty cycle of 0% (always off); FFFF is a duty cycle of 100% (always on). 8000 hex gives a duty cycle of 50%. `PWM-PERIOD` must be specified before using `PWM-OUT`.
`PWM-IN` Makes the counter/timer pin an input, and measures the width of one pulse on that input. Returns a value from 1 to FFFF hex. The counter rate is 2.5

MHz, thus each count is 0.4 usec, and a returned value of 10000 decimal corresponds to 4 msec.

Examples of use:

```
TC0 ON    ( output a high level on the TC0 pin )
TA3 ON?   ( check if TA3 pin, HOME0, is a logic '1' )

DECIMAL 50000 TC1 PWM-PERIOD  ( specify 20 msec period = 50
Hz )
TC1 ACTIVE-HIGH               ( specify active-high output
)
HEX 4000 TC1 PWM-OUT          ( output 25% high, 75% low )
```

PWM I/O Class

These words support the PWM generators of the DSP56F80x. The following PWM outputs are defined as objects:

```
PWMA0   PWMA1   PWMA2   PWMA3   PWMA4   PWMA5
PWMB0   PWMB1   PWMB2   PWMB3   PWMB4   PWMB5
```

For each PWM output, the following methods can be performed:

`ON` Outputs a '1' (high level).
`OFF` Outputs a '0' (low level).
`TOGGLE` Inverts the output level.
`n SET` Stores a T/F value to the pin, e.g., `1 PWMA0 SET`. Any nonzero value is "true."

The following methods can be used to generate PWM signals:

`n PWM-PERIOD` Initializes the PWM output, and specifies its period (frequency). Values from 100 to 7FFF hex are valid. The effective counter frequency is 2.5 MHz; 7FFF hex corresponds to a period of 13.106 msec (76 Hz). `PWM-PERIOD` must be specified before using `PWM-OUT`. ***Note:*** *setting the period for* ***any*** *"A" PWM will affect all six "A" PWMs. Setting the period for* ***any*** *"B" PWM will affect all six "B" PWMs.*

`n PWM-OUT` Outputs a continuous PWM signal with the given duty cycle. Values from 0 to FFFF hex are valid. 0 is a duty cycle of 0% (always off); FFFF is a duty cycle of 100% (always on). 8000 hex gives a duty cycle of 50%. `PWM-PERIOD` must be specified before using `PWM-OUT`.

The following PWM inputs are defined as objects:

```
FAULTA0   FAULTA1   FAULTA2   FAULTA3   ISA0      ISA1
ISA2
FAULTB0   FAULTB1   FAULTB2   FAULTB3   ISB0      ISB1
ISB2
```

For each PWM input, the following methods can be performed:

`GETBIT` Returns pin value (as a bit mask).
`ON?` Returns true if pin is '1' (high level).
`OFF?` Returns true if pin is '0' (low level).

Examples of use:

```
PWMB0 ON  ( output a high level on the PWMB0 pin )
ISA1 ON?  ( check if ISA1 pin is a logic '1' )

DECIMAL 25000 PWMA1 PWM-PERIOD     ( specify 10 msec period
= 100 Hz )
HEX 4000 PWMA1 PWM-OUT        ( output 25% high, 75% low )
```

SPI I/O Class

These words support the SPI port of the DSP56F80x. Only one SPI port is present; it is referenced as object

```
SPI0
```

The following methods can be performed for the SPI port:

`MASTER`	Specifies that the DSP56F80x will act as an SPI Master.
`n BITS`	Specifies the number of bits to be sent by `TX-SPI` and read by `RX-SPI`. Values from 2 to 16 are valid.
`MSB-FIRST`	Specifies that words should be sent and received MSB first.
`LSB-FIRST`	Specifies that words should be sent and received LSB first.
`n MBAUD`	Specifies the bit rate to be used for the SPI port. Four values can be specified: 20 (20 Mbits/sec), 5 (5 Mbits/sec), 2 (2.5 Mbits/sec), and 1 (1.25 Mbits/sec). All other values will be ignored and will leave the baud rate unchanged.
`n TX-SPI`	Transmits one word on the SPI port. This will output 2 to 16 bits on the MOSI pin (Master mode) and generate 16 clocks on the SCLK pin. *This will simultaneously input 2 to 16 bits on the MISO pin (Master mode).*
`RX-SPI`	Receives one word from the SPI port. This word must already have been shifted into the receive shift register; if it has not, RX-SPI will wait for it to be shifted in. *In Master mode, data will only be shifted in when a word is transmitted by TX-SPI. In this mode you should use RX-SPI immediately after TX-SPI to read the data that was received.*

It is acceptable to specify all the SPI parameters after selecting the SPI port. Example of use:

```
SPI0 MASTER 16 BITS MSB-FIRST 5 MBAUD
SPI0 TX-SPI SPI0 RX-SPI
```

The default polarity for the SPI port is CPHA=0, CPOL=1. This means that the SCLK line will be high between words, and that the slave should clock data on the falling edge. (Refer to figure 13-4 in the Motorola DSP56F801-7 Users Manual.)

ADC I/O Class

These words support the A/D converter of the DSP56F80x. The following ADC inputs are defined as objects:

```
ADC0   ADC1   ADC2   ADC3   ADC4   ADC5   ADC6   ADC7
```

Only one method can be used with A/D inputs:

ANALOGIN Reads the A/D input and returns its value. The result is in the range 0-7FF8. (The 12-bit A/D result is left-shifted 3 places.) 7FF8 corresponds to an input of Vref. 0 corresponds to an input of 0 volts.

Example of use:

```
ADC7 ANALOGIN  ( read A/D channel 7, pin AN7 )
```

LOOPINDEX Class

These words support the Looping structure of IsoMax™. The following are defined as objects:

```
LOOPINDEX


LOOPINDEX name          ...to define a loop variable.
```

The following methods can be performed for LOOP INDEX:

MASTER Specifies that the DSP56F80x will act as an SPI Master.
n BITS Specifies the number of bits to be sent by TX-SPI and read by RX-SPI. Values from 2 to 16 are valid.

```
name n START          ...set starting value (default 0)
name n END            ...set ending value (default 1)
name n STEP           ...set increment (default 1)
name COUNT            ...count, and return a truth value
name RESET            ...reset to starting value
name VALUE            ...return the current loop index

Here's the test code that I've used:

\ TESTING CODE
DECIMAL

\ CYCLE expects an object to be named, e.g. FRED CYCLE
LOOPINDEXES
: CYCLE   RESET  BEGIN VALUE . COUNT UNTIL ;

LOOPINDEX FRED   FRED 1 START 10 END 1 STEP
LOOPINDEX WILMA  WILMA 10 START 1 END -1 STEP
```

IsoPod™ HARDWARE FEATURES

- Three On Board LED's
  Red, Yellow, Green
- 16 GPIO lines
  Programmable Edge sensitive interrupts
- Serial Communication Interface (SCI) full-duplex serial channel
  One RS-232
  One RS422/485
  Programmable Baud Rates, 38,400, 19,200, 9600, 4800, 1200
- Serial Peripheral Interface (SPI)
  Full-duplex synchronous operation on four-wire interface
  Master or Slave
- 8-ch 12-bit AD
  Continuous Conversions @ 1.2us (6 ADC cycles)
  Single ended or differential inputs
- 12-channel PWM module
  15-bit counter with programmable resolutions down to 25ns
  Twelve independent outputs,
  or Six complementary pairs of outputs, or combinations
- Eight Timers
  16-bit timers
  Count up/down, Cascadable
- Two Quadrature Decoder
  32-bit position counter
  16-bit position difference register
  16-bit revolution counter
  40MHz count frequency (up to)
- CAN 2.0 A/B module for networking
  Programmable bit rate up to 1Mbit: Multiple boards can be networked (MSCAN)
  Ideal for harsh or noisy environments, like automotive applications
- JTAG port for CPU debugging
  Examine registers, memory, peripherals
  Set breakpoints
  Step or trace instructions
- WatchDog Timer/COP module, Low Voltage Detector for Reset
- Low Voltage, Stop and Wait Modes
- On Board level translation for RS232, RS422, CAN
- On Board Voltage Regulation

CIRCUIT DESCRIPTION

Under construction…

The processor chip contains the vast majority of the circuitry. The remaining support circuitry is described here. The power for the system can be handled several different way, but as the board comes, power will normally be supplied from the VIN pin on J1.

RS-232 Levels Translation

The MAX3221/6/7 converts the 3.3V supply to the voltages necessary to drive the RS-232 interface. Since a typical RS-232 line requires 10 mA of outputs at 10V or more, the MAX3221/6/7 uses about 30 mA from the 3.3V supply. A shutdown is provided, controlled by TD0.

The RS-232 interface allows the processor to be reset by the host computer through manipulation of the ATN line. When the ATN line is low (a logical "1" in RS-232 terms) the processor runs normally. When the ATN line is high (a logical "0" in RS-232 terms) the processor is held in reset.

http://pdfserv.maxim-ic.com/arpdf/MAX3221-MAX3243.pdf

(V2 http://pdfserv.maxim-ic.com/arpdf/MAX3222-MAX3241.pdf)

RS-422/485 Levels Translation

Two MAX3483 buffer the digital signals to RS-422/485 levels. One, U3, always transmits. The other can receive, or transmit. It will normally be used for the receiver in RS-422 double twisted pair communications applications, and the transceiver in RS-485 single twisted pair communications applications. TD1 controls the turn around on U4 allowing RS-485 communications.

http://pdfserv.maxim-ic.com/arpdf/MAX3483-MAX3491.pdf

CAN BUS Levels Translation

A TJA1050 buffers the CAN BUS signal.
http://my.semiconductors.com/acrobat/datasheets/TJA1050_3.pdf

LED's

A 74AC05 drives the on-board LED's. Each LED has a current limiting resistor to the +3.3V supply.
http://www.fairchildsemi.com/ds/74/74AC05.pdf

RESET

A S80728HN Low Voltage Detector asserts reset when the voltage is below operating levels. This prevents brown out runaway, and a power-on-reset function.

http://www.seiko-instruments.de/documents/ic_documents/power_e/s807_e.pdf

POWER SUPPLY

A LM2937 reduces the VIN DC to a regulated 5V. In early versions a 7805C was used. The LM2937 was rated a bit less for current (500 mA Max), but had reverse voltage protection and a low drop out which was more favorable. A drops the 5V to the 3.3V needed for the processor. At full current, 200 mA, these two regulators will get hot. They can provide current to external circuits if care is taken to keep them cool. Each are rated at 1A but will have to have heat sinking added to run there.

http://www.national.com/ds/LM/LM2937.pdf
http://www.national.com/ds/LM/LM3940.pdf

TROUBLE SHOOTING

There are no user serviceable parts on the IsoPod™. If connections are made correctly, operation should follow, or there are serious problems on the board. As always, the first thing to check in case of trouble is checking power and ground are present. Measuring these with a voltmeter can save hours of head scratching from overlooking the obvious. After power and ground, signal connections should be checked next. If the serial cable comes loose, on either end, using your PC to debug your program just won't help. Also, if your terminal program has locked up, you can experience some very "quiet" results. Don't overlook these sources of frustrating delays when looking for a problem. They are easy to check, and will make a monkey of you more times than not, if you ignore them.

One of the great advantages of having an interactive language embedded in a processor, is if communications can be established, then program tools can be built to test operations. If the RS-232 channel is not in use in your application, or if it can be optionally assigned to debugging, talking to the board through the language will provide a wealth of debugging information.

The LED's can be wonderful windows to show operation. This takes some planning in design of the program. A clever user will make good use of these little light. Even if the RS-232 channel is in use in your application and not available for debugging, don't overlook the LED's as a way to follow program execution looking for problems.

The IsoPod™ is designed so no soldering to the board should be required, and the practice of soldering to the board is not recommended. Instead, all signals are brought to connectors. That's one of the reasons it is called a "Pod", it can be plugged in and pulled out as a module.

So, the best trouble shooting technique would be to unplug the IsoPod™ and try to operate it separately with a known good serial cable on power supply.

If the original connections have been tested to assure no out-of-range voltages are present, a second IsoPod™ can then be programmed and plugged into the circuit in question. But don't be too anxious to take this step. If the first IsoPod™ should be burned out, you really want to be sure you know what caused it, before sacrificing another one in the same circuit.

Finally, for advanced users, the JTAG connection can give trace, single step and memory examination information with the use of special debugging hardware. This level of access is beyond the expected average user of the IsoPod™ and will not be addressed in this manual.

IsoPod™ website: http://www.isopod.net

MaxFORTH™ Glossary Reference Page

http://www.ee.ualberta.ca/~rchapman/MFwebsite/V50/Alphabetical/Brief/index.html

This has explanations for the definitions for the procedural language "under" the IsoMax(TM) Finite State Machine language.

Motorola DSP56F805 Users Manual

http://e-www.motorola.com/brdata/PDFDB/docs/DSP56F801-7UM.pdf

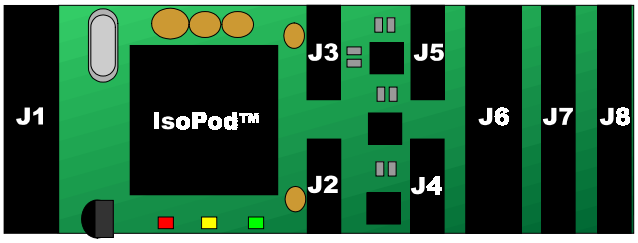
Motorola DSP56F800 Processor Reference Manual

http://e-www.motorola.com/brdata/PDFDB/docs/DSP56800FM.pdf

CONNECTORS V1

The IsoPod™ V1 has 8 connectors. J1, J2, J3, J4, J5, J6, J7, J8 are shown below:

J1	Ser., Power, General Purpose I/O	Serial, Power, Ports PA0 – PA7, PB0 – PB7
J2	JTAG connector	CPU Port, for factory use only
J3	SPI	SCLK, MISO, MOSI, SS, PD0, PD1, PD2, PD3
J4	RS-422/485 Serial Port	-RCV, +RCV, -XMT, +XMT
J5	CAN BUS Network Port	CANL, CANH
J6	Servo Motor Outputs x 12	PWM, V+, GND
J7	Motor Encoder x 2	Quadrature, Fault0, Fault1, Fault2, IS0, IS1, IS2
J8	A/D Various	A/D0 – A/D7, Various

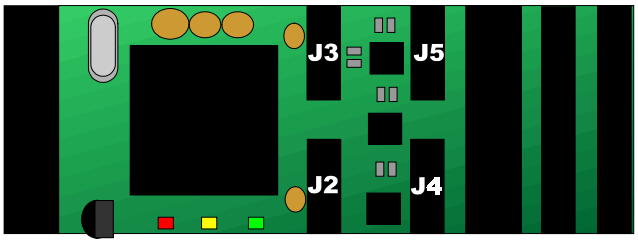


J1 GPIO

+VIN	24	1	SOUT
GND	23	2	SIN
RST’	22	3	ATN’
+5V	21	4	GND
PA0	20	5	PB0
PA1	19	6	PB1
PA2	18	7	PB2
PA3	17	8	PB3
PA4	16	9	PB4
PA5	15	10	PB5
PA6	14	11	PB6
PA7	13	12	PB7

Note: In picture above, Pin 1 is at top left viewing CPU side, with J1 at left. When facing J1 connector, looking straight in, with CPU side to your right, Pin 1 will be at the top right.

This connector pin out and pin numbering scheme is unique to this one instance. Origin of pin out and numbering is to match stamp-like connection pin outs.



Connectors in above “top view, J1-to-left” picture and on page below, have same oriented (pin 1 upper left).

J3 SPI V1

+3V	1	2	GND
PD0	3	4	SCLK
PD1	5	6	MOSI
PD2	7	8	MISO
PD3	9	10	SS’

J5 CAN BUS V1

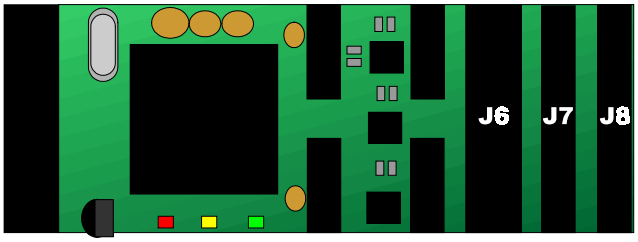
N.C.	1	2	N.C.
CANL	3	4	CANH
N.C.	5	6	GND
N.C.	7	8	N.C.
N.C.	9	10	N.C.

J2 JTAG V1

+3V	1	2	GND
TDI	3	4	GND
TDO	5	6	TMS
TCK	7	8	DE
RESET’	9	10	TRST

J4 RS-422/485 V1

N.C.	1	2	N.C.
+RCV	3	4	-RCV
GND	5	6	GND
-XMT	7	8	+XMT
N.C.	9	10	N.C.



Connectors in above “top view, J1-to-left” picture and on page below, have same oriented (pin 1 upper left).

J6 PWM SERVO OUTPUT V1

	Sig.	+V	GND
PWMB5	1	2	3
PWMB4	4	5	6
PWMB3	7	8	9
PWMB2	10	11	12
PWMB1	13	14	15
PWMB0	16	17	18
PWMA5	19	20	21
PWMA4	22	23	24
PWMA3	25	26	27
PWMA2	28	29	30
PWMA1	31	32	33
PWMA0	34	35	36

J7 Motor Encoder x 2 V1

+5V	1	2	FAULTA0
GND	3	4	FAULTA1
PH A 0	5	6	FAULTA2
PH B 0	7	8	ISA0
IND 0	9	10	ISA1
HM 0	11	12	ISA2
+5V	13	14	FAULTB0
GND	15	16	FAULTB1
PH A 1	17	18	FAULTB2
PH B 1	19	20	ISB0
IND 1	21	22	ISB1
HM 1	23	24	ISB2

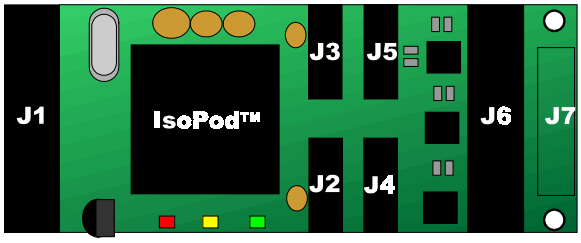
J8 Various V1

ANA0	1	2	+5V
ANA1	3	4	IRQA
ANA2	5	6	IRQB
ANA3	7	8	FAULTB3
ANA4	9	10	FAULTA3
ANA5	11	12	PD5
ANA6	13	14	TC0
ANA7	15	16	TC1
VSSA	17	18	CLKO
VREF	19	20	RSTO
VSS(GND)	21	22	RD'
V+	22	24	WR'

CONNECTORS V2

The IsoPod™ V2 has 7 connectors. J1, J2, J3, J4, J5, J6, J7 are shown below:

J1	Ser., Power, GPI/O	Serial, Power, Ports PA0 – PA7, PB0 – PB7
J2	JTAG connector	CPU Port, for factory use only
J3	A/D	
J4	RS-232/422/485 CAN Bus	-RCV, +RCV, -XMT, +XMT, CANL, CANH
J5	I/O SPI	SCLK, MISO, MOSI, SS, PD0, PD1, PD2, PD3
J6	PWM, Motor Encoder, Timers	PWM, TMRA0-3, TMRB0-3, TMRC0,1 TMRD0-3
J7	Fault & Current Sense	FAULTA0-3, ISA0-2, FAULTB0-3, ISB0-2

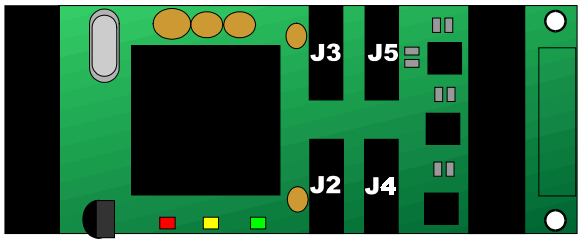


J1 GPIO

+VIN	24	1	SOUT
GND	23	2	SIN
RST’	22	3	ATN’
+5V	21	4	GND
PA0	20	5	PB0
PA1	19	6	PB1
PA2	18	7	PB2
PA3	17	8	PB3
PA4	16	9	PB4
PA5	15	10	PB5
PA6	14	11	PB6
PA7	13	12	PB7

Note: In picture above, Pin 1 is at top left viewing CPU side, with J1 at left. When facing J1 connector, looking straight in, with CPU side to your right, Pin 1 will be at the top right.

This connector pin out and pin numbering scheme is unique to this one instance. Origin of pin out and numbering is to match stamp-like connection pin outs.



Connectors in above “top view, J1-to-left” picture and on page below, have same oriented (pin 1 upper left).

J3 A/D V2

VREF	1	2	VSSA
ANA0	3	4	ANA4
ANA1	5	6	ANA5
ANA2	7	8	ANA6
ANA3	9	10	ANA7

J5 IO/SPI V2

+5V	1	2	GND
+3V	3	4	SCLK
RST0’	5	6	MOSI
PE2	7	8	MISO
PE3	9	10	SS’

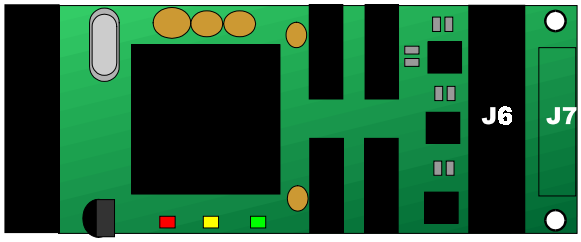
J2 JTAG V2

+3V	1	2	GND
TDI	3	4	GND
TDO	5	6	TMS
TCK	7	8	DE
RESET’	9	10	TRST

J4 RS-232/422/485 CAN BUS V2

	+ XMT	1	2	+5V
	- XMT	3	4	GND
GND	GND	5	6	CANL
SIN1*	- RCV	7	8	GND
SOUT1*	+ RCV	9	10	CANH

* SIN1, SOUT1 RS232 signals



Connectors in above “top view, J1-to-left” picture and on page below, have same oriented (pin 1 upper left).

J6 PWM, Motor Encoder, Timers V2

1	PWMA0	2	+5V	3	+3V
4	PWMA1	5	GND	6	GND
7	PWMA2	8	PH A 0	9	TMRC0
10	PWMA3	11	PH B 0	12	TMRC1
13	PWMA4	14	IND 0	15	IRQA
16	PWMA5	17	HM 0	18	IRQB
19	PWMB0	20	+5V	21	+3V
22	PWMB1	23	GND	24	GND
25	PWMB2	26	PH A 1	27	TMRD0
28	PWMB3	29	PH B 1	30	TMRD1
31	PWMB4	32	IND 1	33	TMRD2
34	PWMB5	35	HM 1	36	TMRD3

J7 Fault & Current Sense V2

FAULTA0	1	2	N.C.
FAULTA1	3	4	ISA0
FAULTA2	5	6	ISA1
FAULTA3	7	8	ISA2
FAULTB0	9	10	ISB0
FAULTB1	11	12	ISB1
FAULTB2	13	14	ISB2
FAULTB3	15	16	N.C.

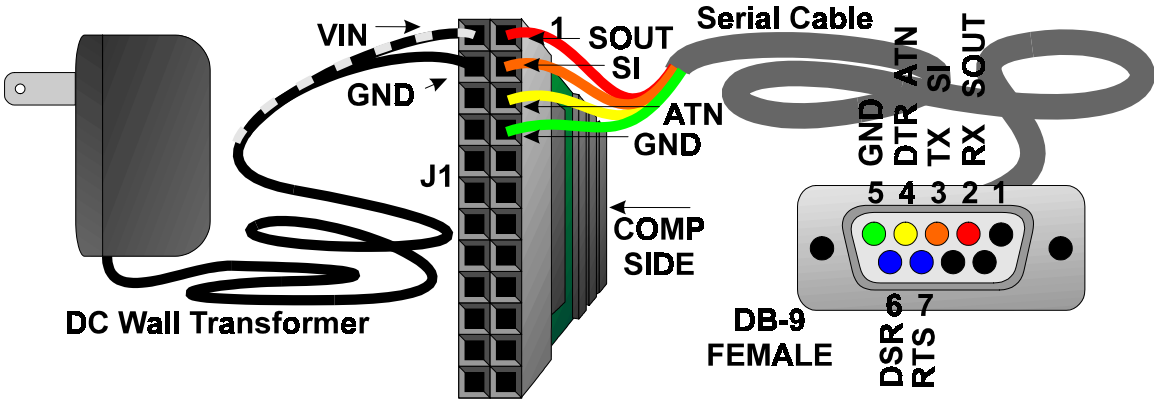
Instructions for Wiring a Serial Cable V1 & V2

Transformer hook up

Black w/Striped White +VIN	24	1	SOUT
Solid Black GND	23	2	SIN
RST'	22	3	ATN'
+5V	21	4	GND
PA0	20	5	PB0
PA1	19	6	PB1
PA2	18	7	PB2
PA3	17	8	PB3
PA4	16	9	PB4
PA5	15	10	PB5
PA6	14	11	PB6
PA7	13	12	PB7

Serial Cable hook up

+VIN	24	1	SOUT RED
GND	23	2	SIN ORANGE
RST'	22	3	ATN'YELLOW
+5V	21	4	GND GREEN
PA0	20	5	PB0
PA1	19	6	PB1
PA2	18	7	PB2
PA3	17	8	PB3
PA4	16	9	PB4
PA5	15	10	PB5
PA6	14	11	PB6
PA7	13	12	PB7



J1 Pin	Preferred Color	DB-9 Pin	DB-25 Pin
1 SOUT	RED	2 RX	2 TX
2 SIN	ORANGE	3 TX	3 RX
3 ATN	YELLOW	4 DTR	20 DTR
4 GND	GREEN	5 GND	7 GND
		6 DSR	6 DSR
		7 RTS	20 RTS

JUMPERS V1

The IsoPod™ has no jumpers. This was a design goal realized. Jumper setting on such a small board, are not very practical so have been avoided. A few sites exist where termination resistors can be added. A few port lines are used to control programmable options on the board.

Port line TD0 controls the RS-232 transmitter shutdown.

Port line TD1 controls the RS-485 transceiver turn-around.

JUMPERS V2

The IsoPod™ has no jumpers. This was a design goal realized. Jumper setting on such a small board, are not very practical so have been avoided. A few sites exist where termination resistors can be added. A few port lines are used to control programmable options on the board.

Port line PD5 controls the RS-232 transmitter shutdown. A pull up resistor normally disenables shutdown, if the port line is inactive.

Port line PD4 controls the RS-232 receiver enable. A pull down resistor normally enables the receivers, if the port line is inactive.

Port line PD3 controls the RS-485 transceiver turn-around. A pull down resistor normally enables the receiver, if the port line is inactive.

Port line PD2 controls the RED LED. The built in pull up in the AC05 makes the LED come on, if the port line is inactive.

Port line PD1 controls the YELLOW LED. The built in pull up in the AC05 makes the LED come on, if the port line is inactive.

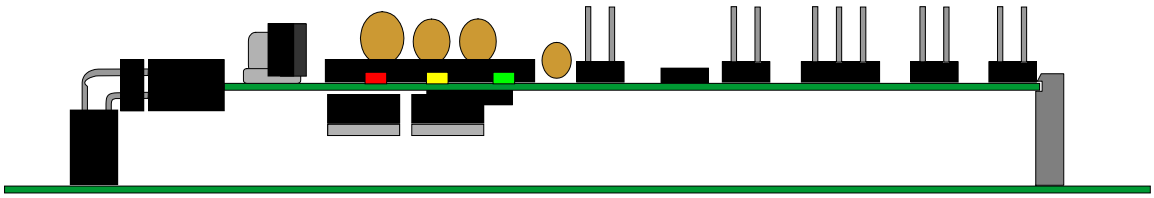
Port line PD0 controls the GREEN LED. The built in pull up in the AC05 makes the LED come on, if the port line is inactive.

Port line PE2 controls the CAN transceiver mode, high-speed mode or silent mode. A pull down resistor normally selects high-speed mode, if the port line is inactive. In the silent mode, the transmitter is disabled. All other IC functions continue to operate. The silent mode is selected by connecting pin S to VCC and can be used to prevent network communication from being blocked, due to a CAN controller which is out of control.

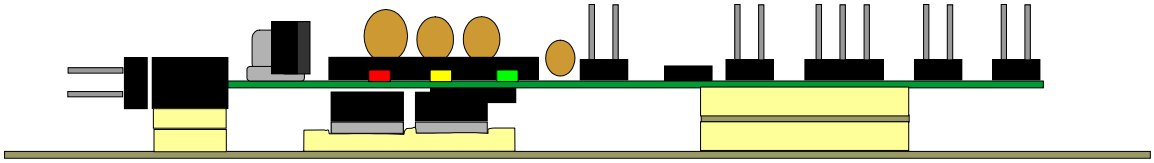
BOARD MOUNTING V1

No mounting holes are provided on the IsoPod™ Board V1, but it may be mounted by:

J1 and supporting clip:

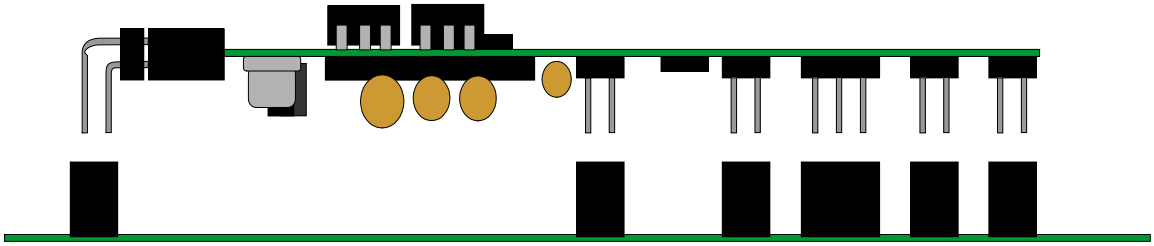


Double sided sticky tape:



Inversion and insertion:

into mating .1” connectors with or without a right angle double male connector on J1



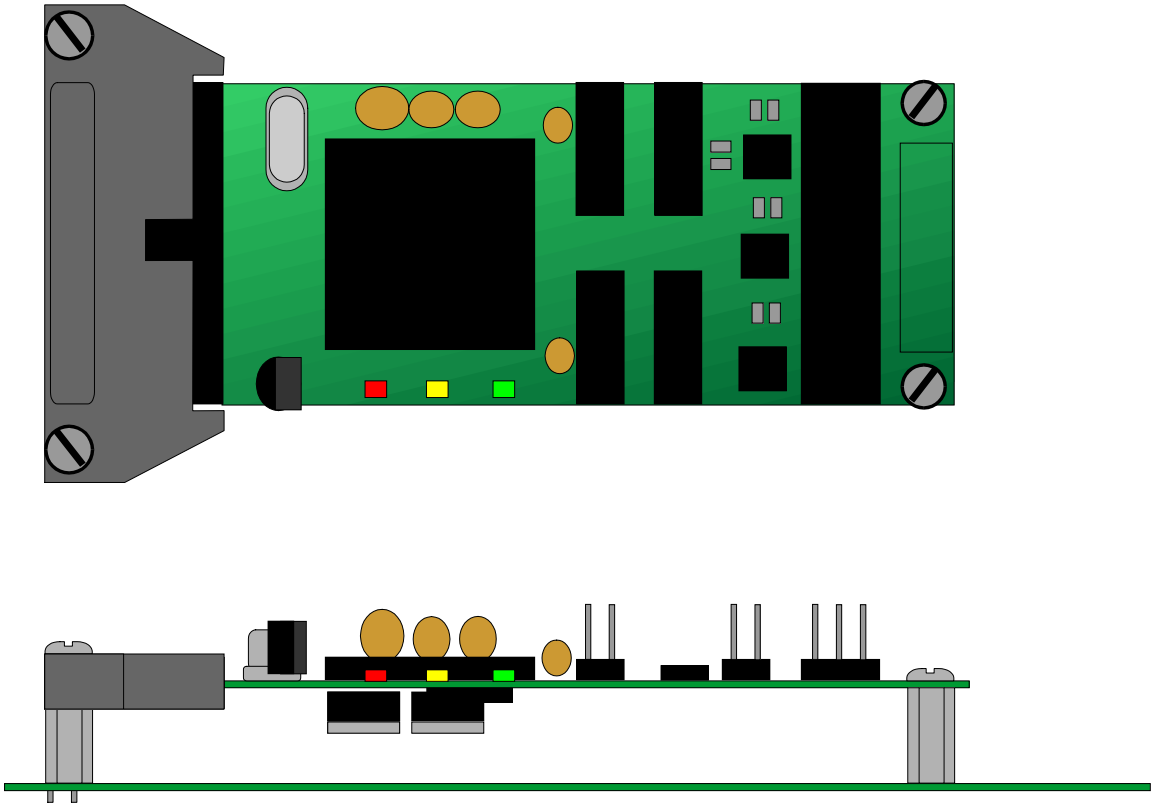
Cable or adapter:

An IDC cable with an IDC male connector can, or an IDC female used with an intermediate double male header, can be ribbon cabled to a similar IDC 24-pin socket header and plugged into an existing stamp-type socket. NMI also manufactures a level, and a right angle adapter for the same purpose.

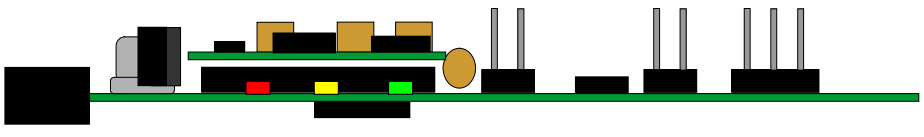
BOARD MOUNTING V2

Two mounting holes are provided on the IsoPod™ Board V2:

J1 Wall Header and supporting standoffs:



V2 Switching Regulator (SR) option:



The Switching Regulator option reduces clearance of analog regulators under board, and eases mounting requirements.

MANUFACTURER

New Micros, Inc.
1601 Chalk Hill Rd.
Dallas, TX 75212

Tel: (214) 339-2204
Fax: (214) 339-1585

Web site: http://www.newmicros.com

This manual: http://www.newmicros.com/store/product_manual/isopod.zip

Email technical questions: nmitech@newmicros.com

Email sales questions: nmisales@newmicros.com

MECHANICAL

Under construction…

Board size is 1.2” x 3”

J1 adds .3” to total board length.

A double male header inserted in J1 will also add length, but since it can be user supplied, only an approximate estimate of .3” can be suggested.

ELECTRICAL

The total draw for the IsoPod™ under maximum speed is approximately 200 mA.

Sleeping or slowing the processor can substantially reduce current consumption.

The TD0 signal can shut down the RS-232 converter, saving about 30 mA, when not used for transmission, if the receiving unit will not sense this as noise.

The TD1 signal can shut down the RS-485 transceiver, U4, saving about 10 mA, when not used for transmission, if the other RS-485 receiving units will not sense this as noise. The other RS-485 transceiver, U3, cannot be shut down, but can be left uninstalled by arrangement with the factory.

Each digital pin is capable of sinking 4 mA and sourcing –4 mA. Each LED draws 1.2 mA when lit.

Absolute Maximum Ratings

Characteristic	Symbol	Min	Max	Unit
Supply voltage	V_{DD}	V_{SS} – 0.3	V_{SS} + 4.0	V
All other input voltages, excluding Analog inputs	V_{IN}	V_{SS} – 0.3	V_{SS} + 5.5V	V
Analog Inputs ANAx, V_{REF}	V_{IN}	V_{SS} – 0.3	V_{DDA} + 0.3V	V
Current drain per pin excluding V_{DD}, V_{SS}, PWM outputs, TCS, V_{PP}, V_{DDA}, V_{SSA}	I	—	10	mA
Current drain per pin for PWM outputs	I	—	20	mA
Junction temperature	T_J	—	150	°C
Storage temperature range	T_{STG}	-55	150	°C

Recommended Operating Conditions

Characteristic	Symbol	Min	Max	Unit
Supply voltage	V_{DD}	3.0	3.6	V
Ambient operating temperature	T_A	-40	85	°C

DC Electrical Characteristics

Operating Conditions: $V_{SS} = V_{SSA} = 0$ V, $V_{DD} = V_{DDA} = 3.0$–3.6 V, $T_A = –40°$ to $+85°C$, $C_L \leq 50$ pF, $f_{op} = 80$ MHz

Characteristic	Symbol	Min	Typ	Max	Unit
Input high voltage	V_{IH}	2.0	—	5.5	V
Input low voltage	V_{IL}	-0.3	—	0.8	V
Input current low (pullups/pulldowns disabled)	I_{IL}	-1	—	1	μA
Input current high (pullups/pulldowns disabled)	I_{IH}	-1	—	1	μA
Typical pullup or pulldown resistance	R_{PU}, R_{PD}	—	30	—	KΩ
Input/output tri-state current	low I_{OZL}	-10	—	10	μA
Input/output tri-state current	low I_{OZH}	-10	—	10	μA
Output High Voltage (at IOH)	V_{OH}	V_{DD} – 0.7	—	—	V
Output Low Voltage (at IOL)	V_{OL}	—	—	0.4	V
Output High Current	I_{OH}	—	—	-4	mA
Output Low Current	I_{OL}	—	—	4	mA
Input capacitance	C_{IN}	—	8	—	pF
Output capacitance	C_{OUT}	—	12	—	pF
PWM pin output source current 1	I_{OHP}	—	—	-10	mA
PWM pin output sink current 2	I_{OLP}	—	—	16	mA
Total supply current	I_{DDT} 3				
Run 4		—	126	162	mA
Wait 5		—	72	98	mA
Stop		—	60	84	mA
Low Voltage Interrupt 6	V_{EI}	2.4	2.7	2.9	V
Power on Reset 7	V_{POR}	—	1.7	2.0	V

1. PWM pin output source current measured with 50% duty cycle.

2. PWM pin output sink current measured with 50% duty cycle.

3. $I_{DDT} = I_{DD} + I_{DDA}$ (Total supply current for $V_{DD} + V_{DDA}$)

4. Run (operating) I_{DD} measured using 8MHz clock source. All inputs 0.2V from rail; outputs unloaded. All ports configured as inputs; measured with all modules enabled.

5. Wait I_{DD} measured using external square wave clock source ($f_{osc} = 8$ MHz) into XTAL; all inputs 0.2V from rail; no DC loads; less than 50 pF on all outputs. $C_L = 20$ pF on EXTAL; all ports configured as inputs; EXTAL capacitance linearly affects wait I_{DD}; measured with PLL enabled.

6. Low voltage interrupt monitors the V_{DDA} supply. When V_{DDA} drops below V_{EI} value, an interrupt is generated. For correct operation, set V_{DDA}=V_{DD}. Functionality of the device is guaranteed under transient conditions when V_{DDA}>V_{EI}.

7. Power-on reset occurs whenever the internally regulated 2.5V digital supply drops below V_{POR}. While power is ramping up, this signal remains active for as long as the internal 2.5V supply is below 1.5V no matter how long the ramp up rate is. The internally regulated voltage is typically 100 mV less than V_{DD} during ramp up until 2.5V is reached, at which time it self regulates.

NMITerm

Provided Windows terminal program from New Micros, Inc. Usually provided in a ZIP. Un ZIP in a subdirectory, such as C:\NMITerm. To start the program: click, or double click, the program icon.

NMITerm is a simple Windows-based communications package designed for program development on serial port based embedded controllers. It runs under Windows.

NMITerm provides:

1. Support for COM1 through COM16.
2. Baud rates from 110 through 256000.
3. Control over RTS and DTR lines.
4. Capture files, which record all terminal activity to disk.
5. Scroll-back buffer, editable and savable as a file.
6. On-line Programmer's Editor.
7. File downloader.
8. Programmable function keys.

Quick start commands:

1. Baud: default 9600
2. DTR On/Off : ALT+T
3. Download: ALT+D

For further information use the F1 Help screen.

This program can be downloaded from:

http://www.newmicros.com/download/NMITerm.zip

MaxTerm

Provided DOS terminal program from New Micros, Inc. Usually provided in a ZIP. Un ZIP in a subdirectory, such as C:\MAXTERM. To start the program: click, or double click, the program icon.


Maxterm.ico

MaxTerm is a simple DOS-based communications package designed for program development on serial port based embedded controllers. It can run under stand-alone DOS or in a DOS session under Windows.

MaxTerm provides:

1. Support for COM1 through COM4.
2. Baud rates from 300 through 38400.
3. Control over RTS and DTR lines.
4. Capture files, which record all terminal activity to disk.
5. 32K scroll-back buffer, editable and savable as a file.
6. On-line Interactive Programmer's Editor (OPIE).
7. File downloader.
8. Programmable function keys.
9. Received character monitor, which displays all data in HEX.

Quick start commands:

4. Set comport: ALT+1 or ALT+2 It does not support com3 & 4.
5. Baud: default 9600
6. DTR On/Off : ALT+T
7. Download: ALT+D
8. PACING: ALT+P (IsoMax default decimal 10)

For further information use the Help screen (ALT-H) or the program documentation.

```
                MAXTERM Help
alt-B Change baud rate               alt-M Character monitor mode
alt-C Open (or close) capture file   alt-O Toggle sounds
alt-D Download a file (all text)     alt-P Change line pace char
alt-E Edit a file (Split screen)     alt-R Toggle RTS
alt-F Edit function keys             alt-S Unsplit the screen
alt-H Help                           alt-T Toggle DTR
alt-I Program Information            alt-U Change colors
alt-K Toggle redefinition catcher    alt-W Wipe the screen
alt-L Open scrollback log            alt-X Exit
alt-1 (2 3 4) Select Com port        alt-Z Download a file (no fat)
f1-f10 Programmable function keys    f12   Re-enter OPIE

Status line mode indicators: r = rts, d = dtr, L = log file, S =
sounds, K = redefinition, P = line pacing active
```

HyperTerminal

Usually provided in Programs/Accessories/Communications/HyperTerminal. If not present, it can be loaded from the Windows installation disk. Use “Add/Remove Software” feature in Settings/Control Panel, choose Windows Setup, choose Communications, click on Hyperterm, then Okay and Okay. Follow any instructions to add additional features to windows.


Hypertrm.exe

C:\Program Files\Accessories\HyperTerminal

Run HyperTerminal, select an icon that pleases you and give the new connection a name, such as ISOPOD. Now in the “Connect To” dialog box, in the bottom “Connect Using” line, select the communications port you wish to use, with Direct Comm1, Direct Comm2, Direct Comm3, Direct Comm4 as appropriate, then Okay. In the COMMx Dialog box which follows set up the port as follows: Bits per second: 9600 , Data bits: 8, Parity: None, Flow Control: None, then Okay.

The ATN signal must be unconnected when using this program. There is no option to remotely set and reset the board using the DTR line with this program.

REFERENCE

Decimal - Hex - ASCII Chart

DEC	HEX	Char	Function
000	00	NUL	Null
001	01	SOH	Start of heading
002	02	STX	Start of text
003	03	ETX	End of text
004	04	EOT	End of transmit
005	05	ENQ	Enquiry
006	06	ACK	Acknowledge
007	07	BEL	Bell
008	08	BS	Back Space
009	09	HT	Horizontal Tab
010	0A	LF	Line Feed
011	0B	VT	Vertical Tab
012	0C	FF	Form Feed
013	0D	CR	Carriage Return
014	0E	SO	Shift Out
015	0F	SI	Shift In
016	10	DLE	Data Line Escape
017	11	DC1	Device Control 1
018	12	DC2	Device Control 2
019	13	DC3	Device Control 3
020	14	DC4	Device Control 4
021	15	NAK	Non Acknowledge
022	16	SYN	Synchronous Idle
023	17	ETB	End Transmit Block
024	18	CAN	Cancel
025	19	EM	End of Medium
026	1A	SUB	Substitute
027	1B	ESC	Escape
028	1C	FS	File Separator
029	1D	GS	Group Separator
030	1E	RS	Record Separator
031	1F	US	Unit Separator

032	20	Space	056	38	8	080	50	P	104	68	h
033	21	!	057	39	9	081	51	Q	105	69	I
034	22	"	058	3A	:	082	52	R	106	6A	J
035	23	#	059	3B	;	083	53	S	107	6B	K
036	24	$	060	3C	<	084	54	T	108	6C	L
037	25	%	061	3D	=	085	55	U	109	6D	M
038	26	&	062	3E	>	086	56	V	110	6E	N
039	27	'	063	3F	?	087	57	W	111	6F	O
040	28	(	064	40	@	088	58	X	112	70	P
041	29	)	065	41	A	089	59	Y	113	71	Q
042	2A	*	066	42	B	090	5A	Z	114	72	R
043	2B	+	067	43	C	091	5B	[	115	73	S
044	2C	,	068	44	D	092	5C	\	116	74	T
045	2D	-	069	45	E	093	5D	]	117	75	U
046	2E	.	070	46	F	094	5E	^	118	76	V
047	2F	/	071	47	G	095	5F	_	119	77	W
048	30	0	072	48	H	096	60	`	120	78	X
049	31	1	073	49	I	097	61	a	121	79	Y
050	32	2	074	4A	J	098	62	b	122	7A	Z
051	33	3	075	4B	K	099	63	c	123	7B	{
052	34	4	076	4C	L	100	64	d	124	7C	|
053	35	5	077	4D	M	101	65	e	125	7D	}
054	36	6	078	4E	N	102	66	f	126	7E	~
055	37	7	079	4F	O	103	67	g	127	7F	DEL

ASCII Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	I	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

More on ASCII on another web site: http://www.jimprice.com/jim-asc.htm

GLOSSARY

Under construction…

.1” double and triple row connectors
24-pin socket
74AC05
9600 8N1
A/D
adapter
ASCII
CAN BUS
Caps
carrier board
computer “pod”
computing and control function
communications channel
communications settings
COMM2
COMM3
COMM4
controller
controller interface board
dedicated computer
deeply embedded
double male right angle connector
double sided sticky tape
embedded
embedded tasks
female
hand-crimped wires
headers
high-density connectors
High-Level-Language
HyperTerminal
IDC headers and ribbon cable
interactive
IsoMax™
IsoPod™
language
Levels Translation
LED
LM3940
LM78L05

Low Voltage Detector
male
MaxTerm
mating force of the connectors

Mealy, G. H. State machine pioneer, wrote "A Method for Synthesizing Sequential Circuits," Bell System Tech. J. vol 34, pp. 1045 –1079, September 1955

mobile robot

Moore, E. F. State machine pioneer, wrote "Gedanken-experiments on Sequential Machines," pp 129 – 153, Automata Studies, Annals of Mathematical Studies, no. 34, Princeton University Press, Princeton, N. J., 1956

Multitasking
PCB board
PWM
PWM connectors
Power Supply
Programming environment
prototyping
RS-232
RS-422
RS-485
R/C Servo motor
real time applications.
real time control
registers
RESET
Resistor
S80728HN
SCI
SPI
serial cable
"stamp-type" controller
stand-alone computer board
TJA1050
terminal program
upgrade an existing application.
Virtually Parallel Machine Architecture™ (VPMA)
wall transformer

INDEX

Under construction…