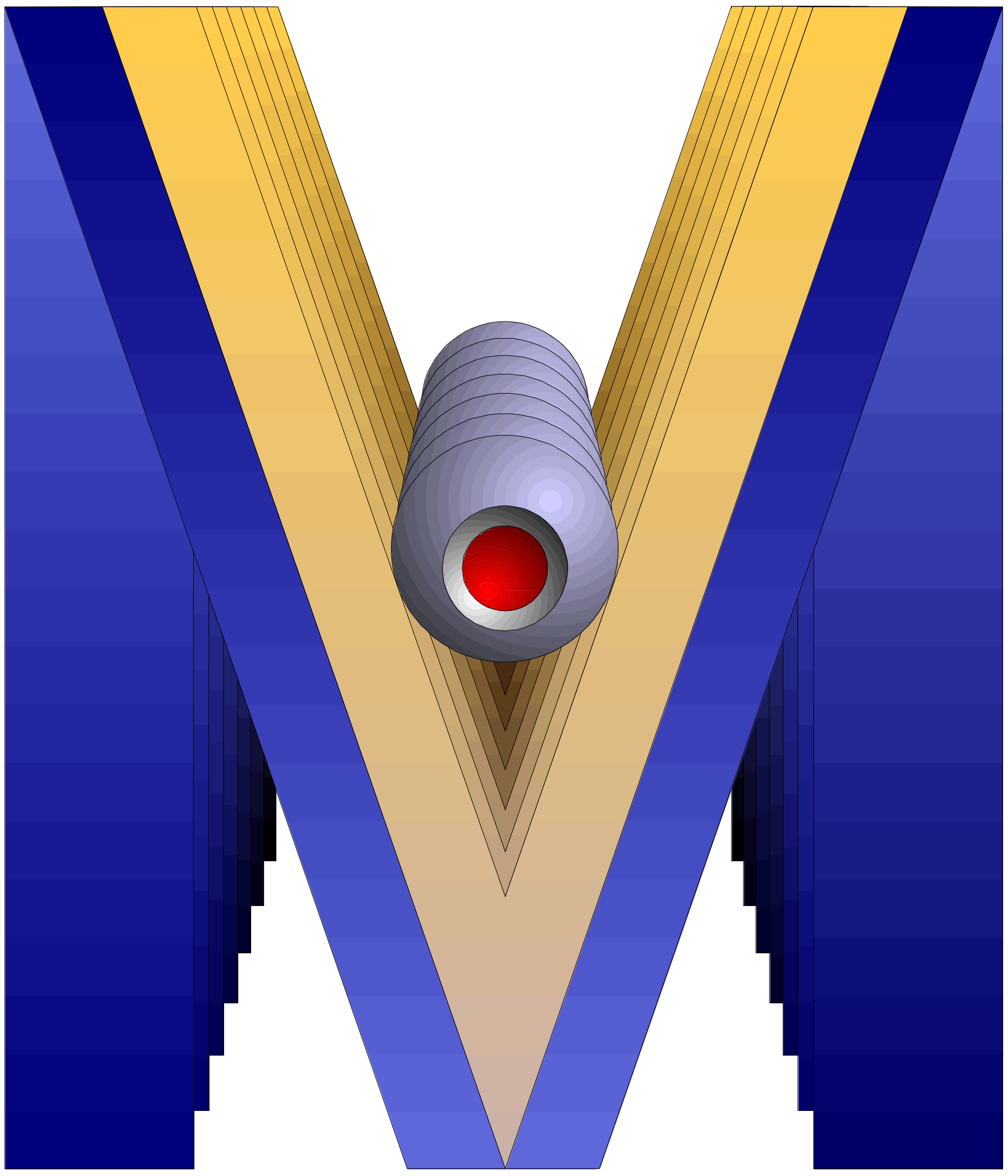
IsoPod™ Users Manual

Virtually Parallel



Machine Architecture

1. 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.

2. Copyrights and Trademarks

Copyright © 2002 by New Micros, Inc. All rights reserved. IsoPod™, IsoMax™ and Virtually Parallel Machine Architecture™ are trademarks of New Micros, Inc. Windows is a registered trademark of Microsoft Corporation. 1-wire is a registered trademark of Dallas Semiconductor. Other brand and product names are trademarks or registered trademarks of their respective holders.

3. 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.

4. 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.

5. 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.

6. GETTING STARTED

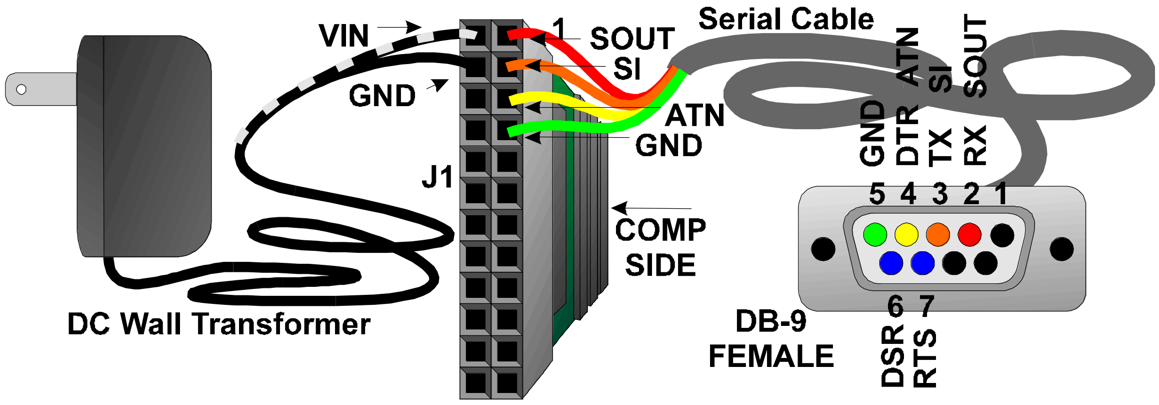
Thank you for buying the IsoPod™. We hope you will find the IsoPod™ to be the incredibly useful small controller board we intended it to be, and easy to use as possible.

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

That's okay. We understand.

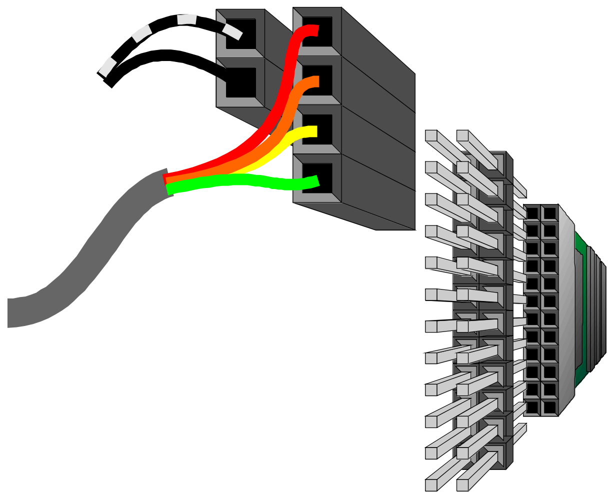
Let's skip the features and the tour and discussion of Virtually Parallel Machine Architecture™ (VPMA) and get right to the operation. Those points can come later. Once we've got communications, then we can make some lights blink and know for sure we're in business. Let's make this "pod" talk to us!

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.



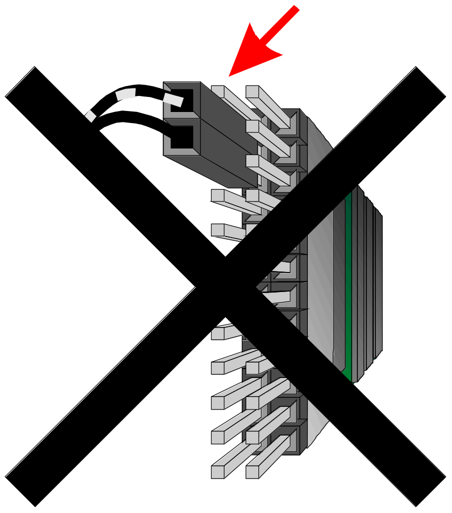
These connections are all made on a few pins of J1, which is a female .1" dual row connector. Download from http://www.newmicros.com/store/product_manual/isopod.zip the manual and read the rest if you haven't yet.

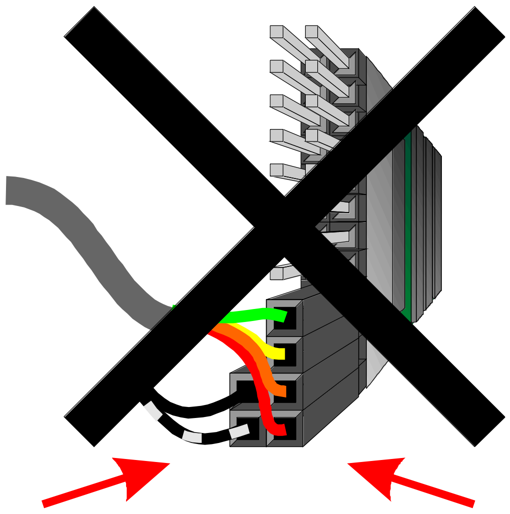
Generally, an intermediate double male header strip will be used to mate from J1 to the Wall transformer single row female connector, and to the Serial Cable single row female connector.



(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.

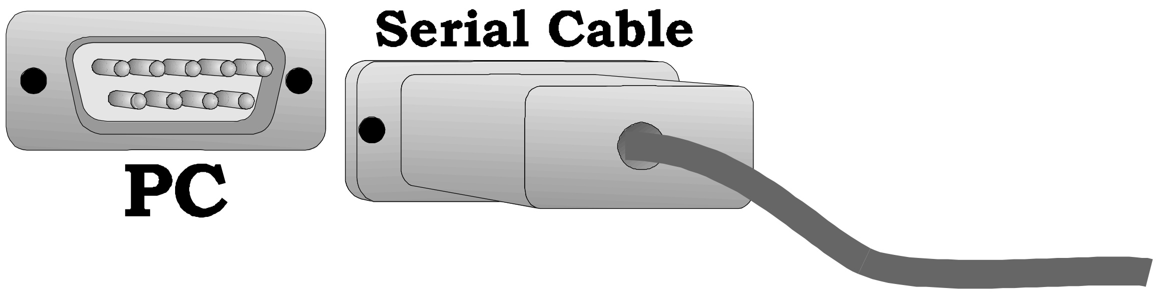




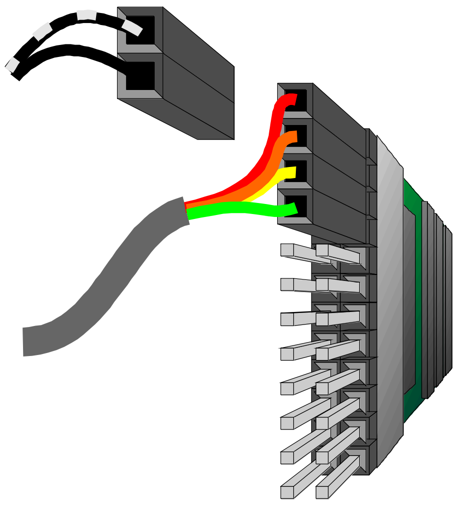
Once you have your serial cable and connectors, and wall transformer and connectors, ready, follow these steps.

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.)

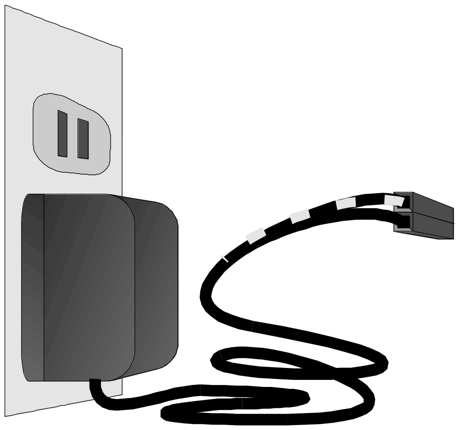
Hook the computer end of the serial cable (usually a DB-9 connector, but may be a DB-25, or other, on older PC's) to the PC's communication channel selected in the terminal program.



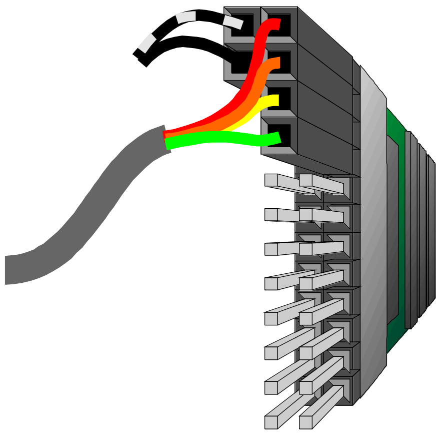
Now hook the IsoPod™ end of the serial cable to the IsoPod™ with connections as shown in the instructions. See the illustration here:



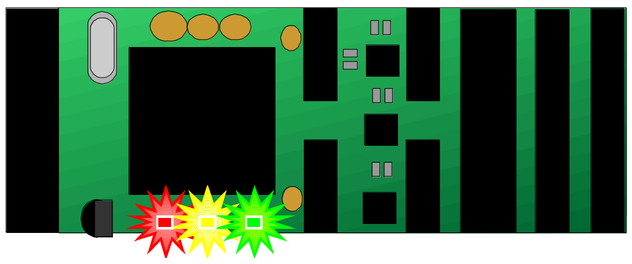
Plug the wall transformer into the wall, but do not plug it into the board yet.



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).

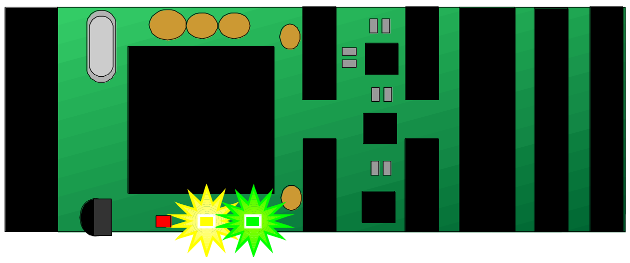
If the LED's don't light, and the screen doesn't show the message, unplug the power to the IsoPod™. Go back through the instructions again. Check the power connections, particularly for polarity. (This is the most dangerous error to your board.) If the LED's come on but there is no communication, check 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™.

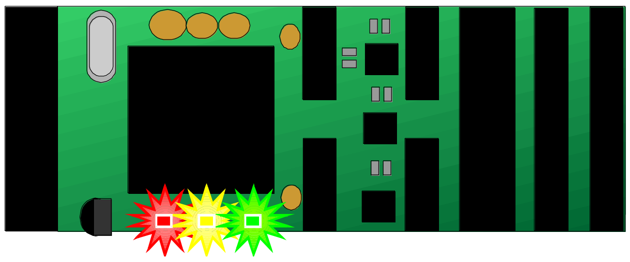
Now let's blink the LED's. Port lines control the LED's. Type:

```
REDLED OFF
```



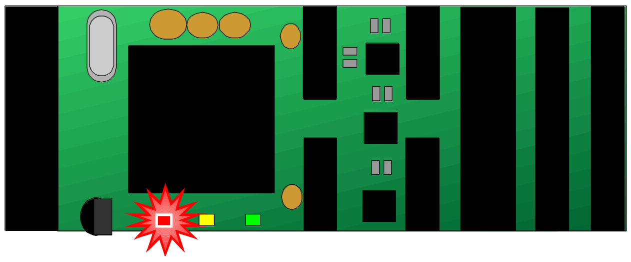
To turn it back on type:

```
REDLED ON
```

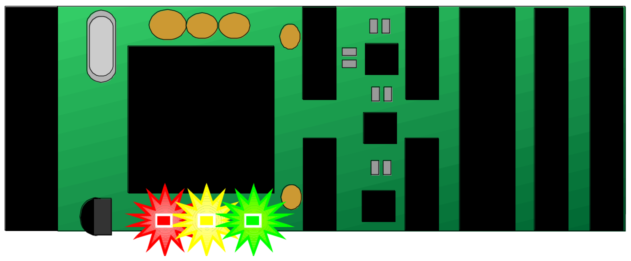


Now let's use the Yellow and Green LED's. Type:

```
YELLED OFF GRNLED OFF
```



```
To turn it back on type:

 YELLED ON GRNLED ON
```



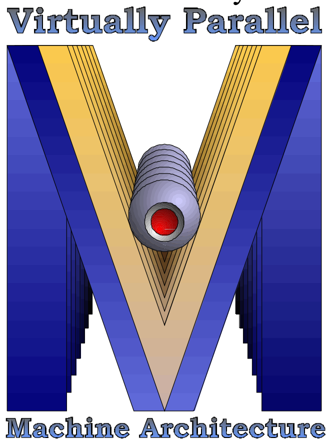
So. Now you should have a good feeling because you can tell your IsoPod™ is working. It's time for an overview of what your IsoPod™ has for features.

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 V0.5 is being shipped, which has support words for many of the built in hardware functions, and V0.6 is already planned which will add more I/O functions. 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.

7. INTRODUCTION

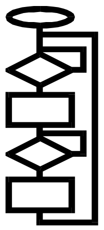
Okay. We should be running. Back to the basics.

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?

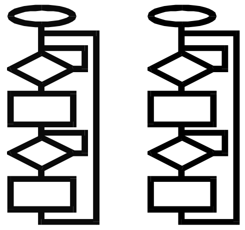


The answer is very important. The neatest thing about the IsoPod™ is software giving Virtually Parallel Machine Architecture!

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.

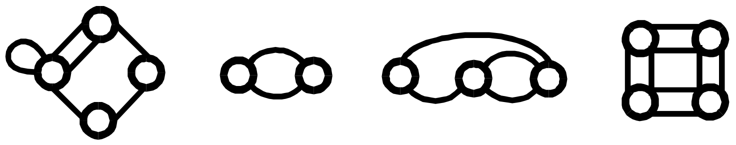


Getting two things done at the same time gets tricky. Add a few more things concurrently competing for processor attention, and most projects start running into serious trouble. Much beyond that, and only the best programmers can weave a program together running many tasks in one application.

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 losing 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.

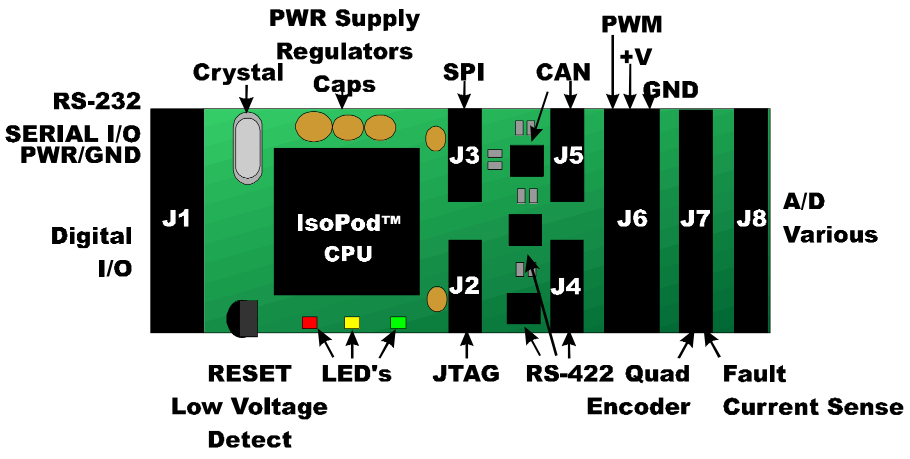
Programming means create a new processor task fashioned as a machine, and debug it interactively in the foreground. When satisfied with performance, you install the new machine in a chain of machines. The machine chain becomes a background feature of the IsoPod™ until you remove it or replace it.

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 cable 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.

8. *QUICK 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.

9. ISOSTRUCTURE

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

Consider this example. Let's say you must hire a night watchman at a dam. Now things run pretty slowly at your dam, so there are four or five critical things which must be monitored, and they need to be adjusted of within half an hour to 45 minutes of getting out of whack, or they'll become critical. So what do you do? You train the night watchman to make rounds every 15 minutes. As long as he gets to all the things that must be checked and adjusted within the 15 minutes, everything is safe.

He's probably fast enough the round will only take a very short amount of the 15 minutes, and he can go eat donuts and drink coffee with the rest of his time. As long as he gets out there and checks everything, sees what conditions are out of whack, and takes corrective action, then moves on, every 15 minutes, it's all fine.

But if the watchman sees one thing go out of whack, and adjusts it and waits there for it to come back into range, (which could take as long to come back as it did to go out) what happens to the other 4 things? Maybe nothing. Or maybe they go out of whack too, and he doesn't get there for an hour because he's been focused on the one thing he first saw was wrong.

If you've got single focus watchman, what do you have to do? You have to have multiple watchman, each doing a single task. Or you have to get an executive who interrupts the single-minded watchman and transports him from check point to check point.

Now while the need for the watchman to keep moving is a simple management issue, the same obviousness is not obvious in software. Why? The structures in most languages discourage anything but spinning on a condition until it clears. Like the watchman fixated on the one problem and stopping his rounds, backwards branches in languages allow a program to become stuck on some bit until it changes.

There are several choices.
1) Only do one thing and settle for that, hire one single-minded watchman for each control (multiprocessing).
2) Hire one single-minded watchman, and hire an executive to interrupt him if he becomes fixated, and move him along (multitasking).
3) Hire one watchman who isn't so single-minded and can never be allowed to stop moving along on his rounds. (Isostructure)

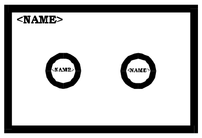
10. ISOMAX PROGRAMMING

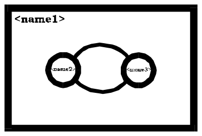
IsoMax is a programming language based on Finite State Machine (FSM) concepts applied to software, with a procedural language (derived from Forth) underneath it. The closest description to the FSM construction type is a “One-Hot” Mealy type of Timer Augmented Finite State Machines. More on these concepts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

11. 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 machine	`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-STATE    CONDITION     CAUSES     THEN-STATE     TO-HAPPEN
    \           /\           /\           /\              /
```

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

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

So here you have learned a bit of the syntax of IsoMax™. Machines are defined, states appended. The transitions are laid out in a pattern, with certain words surrounding others. Procedural parts are inserted in the transitions between the standard clauses.

The syntax is very loose compared to some languages. What is important is the order or sequence these words come in. Whether they occur on one line or many lines, with one space or many spaces between them doesn't matter. Only the order is important.

12. 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.

13. 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.

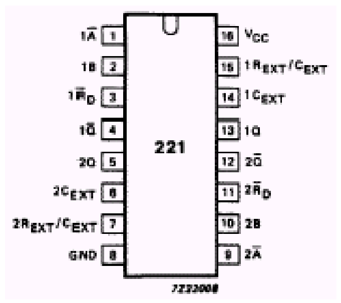


Fig.1 Pin configuration.

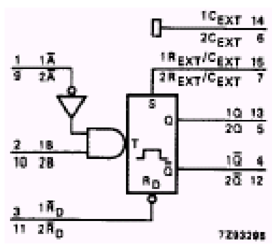


Fig.2 Logic symbol.

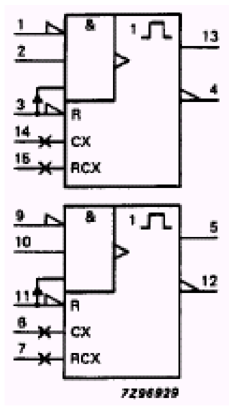
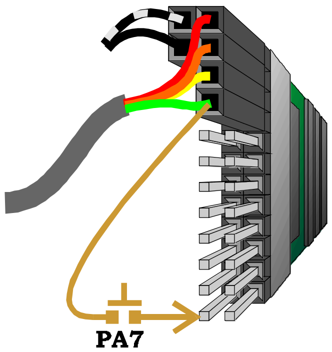


Fig.3 IEC logic symbol.

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.)

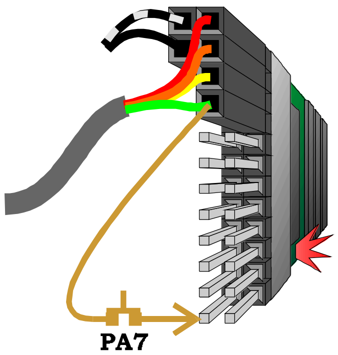
Now if you want, type these lines shown above in. (If you are reading this manual electronically, you should be able to highlight the text on screen and copy the text to the clipboard with Cntl-C. Then you may be able to paste into your terminal program. On MaxTerm, the command to down load the clipboard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



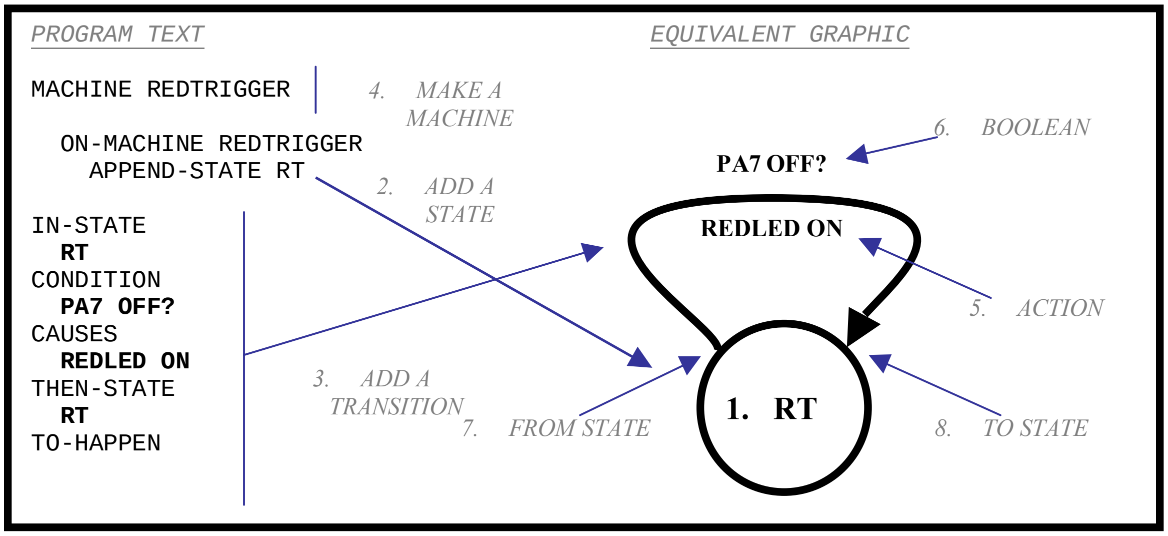
Ground PA7 with a wire or press the push button, and see the red LED come on. Remove the ground or release the push button. The red LED does not go back off. The program is still running, even though all visible changes end at that point. To see that, we'll need to manually reset the LED off so we can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



```
MACHINE REDTRIGGER

  ON-MACHINE REDTRIGGER
    APPEND-STATE RT

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

In this example, the first program line, we 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:

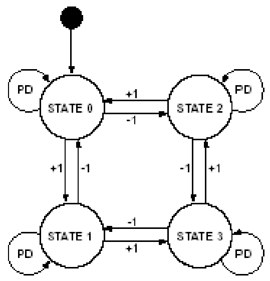
```
IN-STATE RT CONDITION PA7 OFF? CAUSES REDLED ON THEN-STATE RT TO-HAPPEN
```

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.)

14. 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

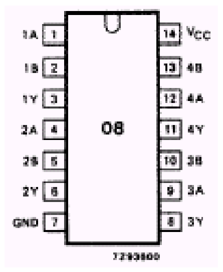


Fig.1 Pin configuration.

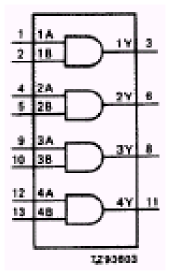


Fig.2 Logic symbol.

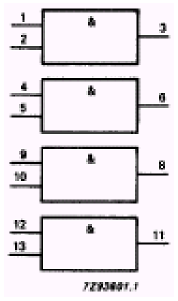


Fig.3 IEC logic symbol.

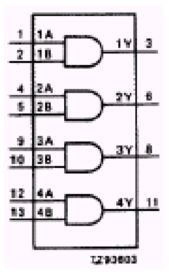


Fig.4 Functional diagram.

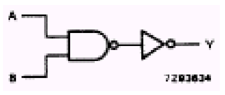


Fig.5 HC logic diagram (one gate).

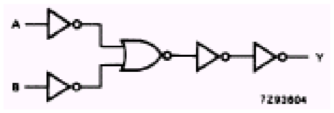


Fig.6 HCT logic diagram (one gate).

FUNCTION TABLE

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

Note

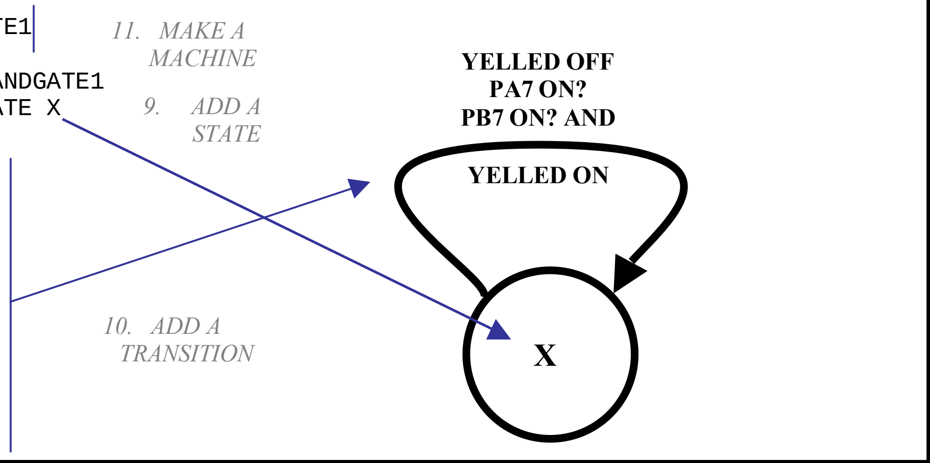
1. H = HIGH voltage level
   L = LOW voltage level

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".

PROGRAM TEXT

```
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
```

EQUIVALENT GRAPHIC



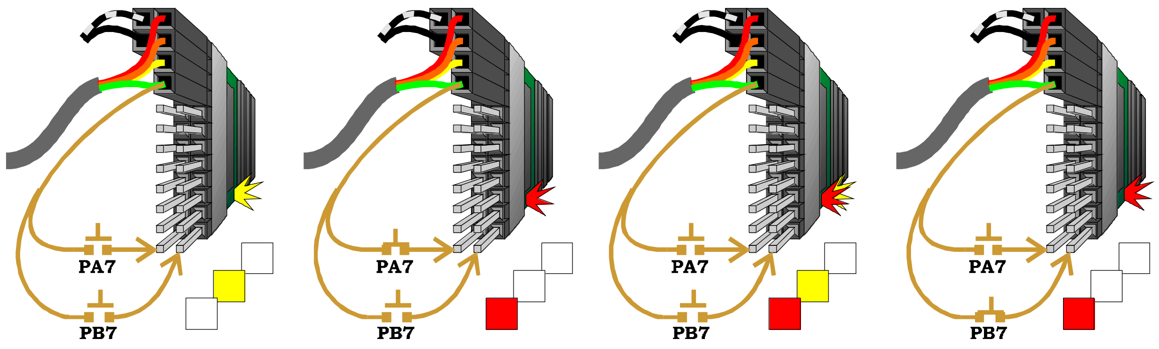
(Now it is worth mentioning, the example is a bit contrived. When you try to make a state machine too simple, you wind up stretching things you shouldn't. This example could have acted exactly like an AND gate if two transitions were used, rather than just one. Instead, a "trick" was used to turn the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

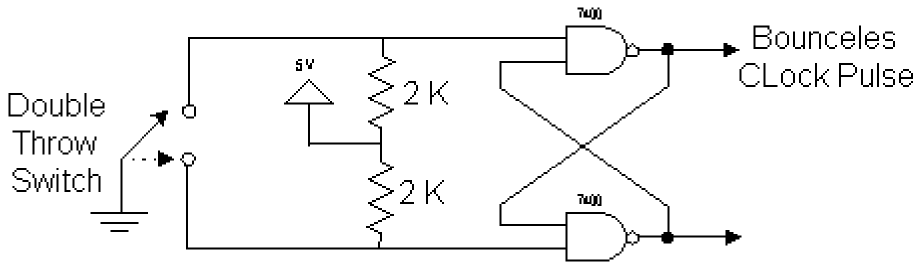
15. BOUNCELESS

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".



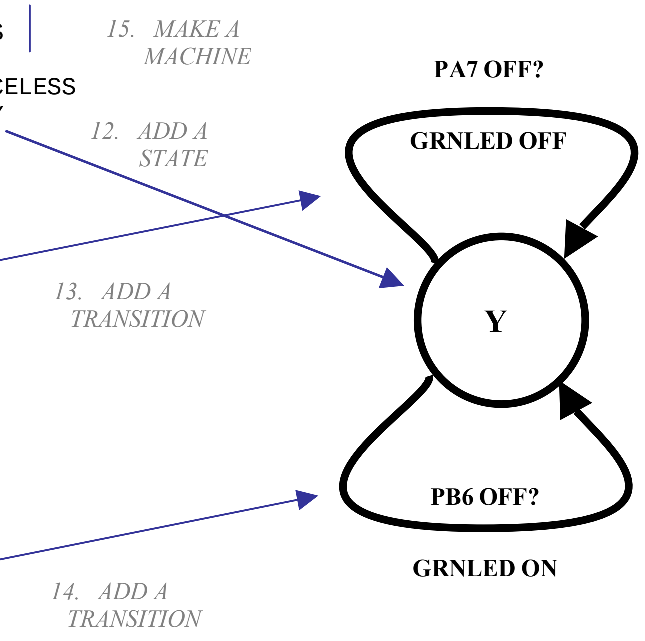
Bounceless switches filter out any noise on their input buttons, and give crisp, one-edge output signals. They do this by toggling state when an input first becomes active, and remaining in that state. If you are familiar with hardware, you might recognize the two gates feed back on each other as a very 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.

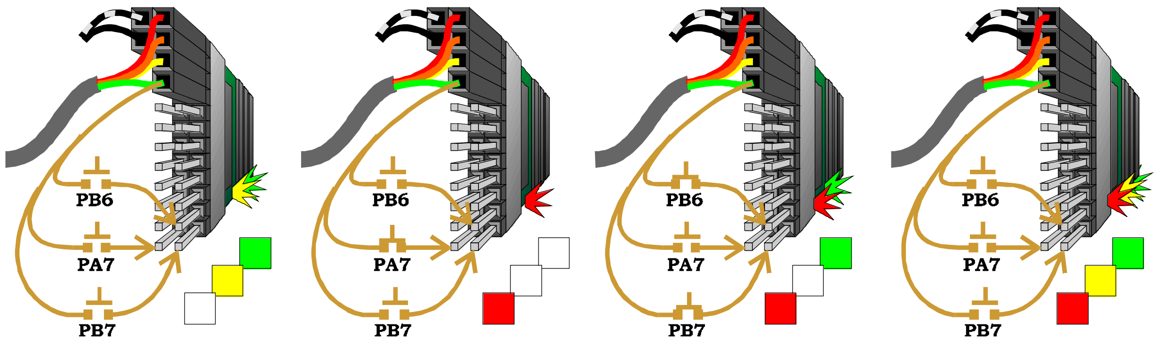
PROGRAM TEXT

```
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
```

EQUIVALENT GRAPHIC



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.

16. 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 by 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.

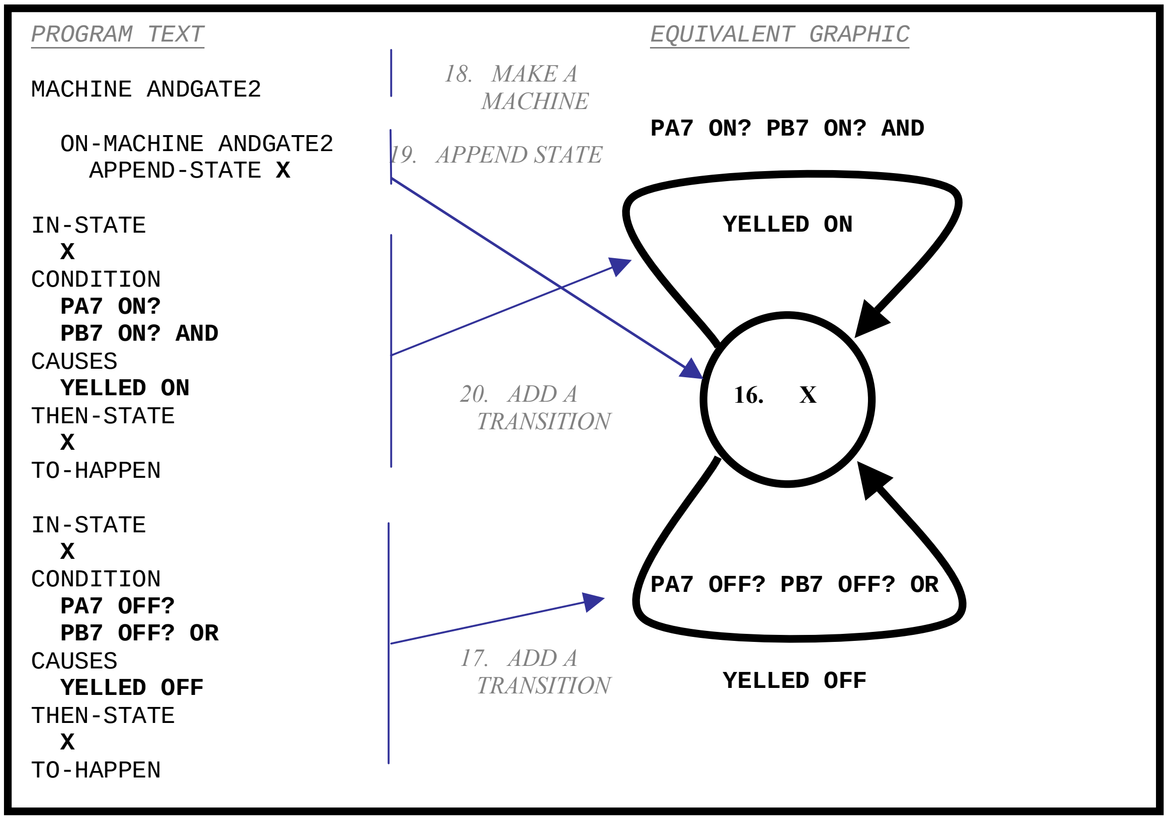
17. MULTIPLE STATES/MULTIPLE TRANSITIONS

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.

18. 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
```



```
MACHINE ANDGATE2

  ON-MACHINE ANDGATE2
    APPEND-STATE X

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

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

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

19. 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". One 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.

20. 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.

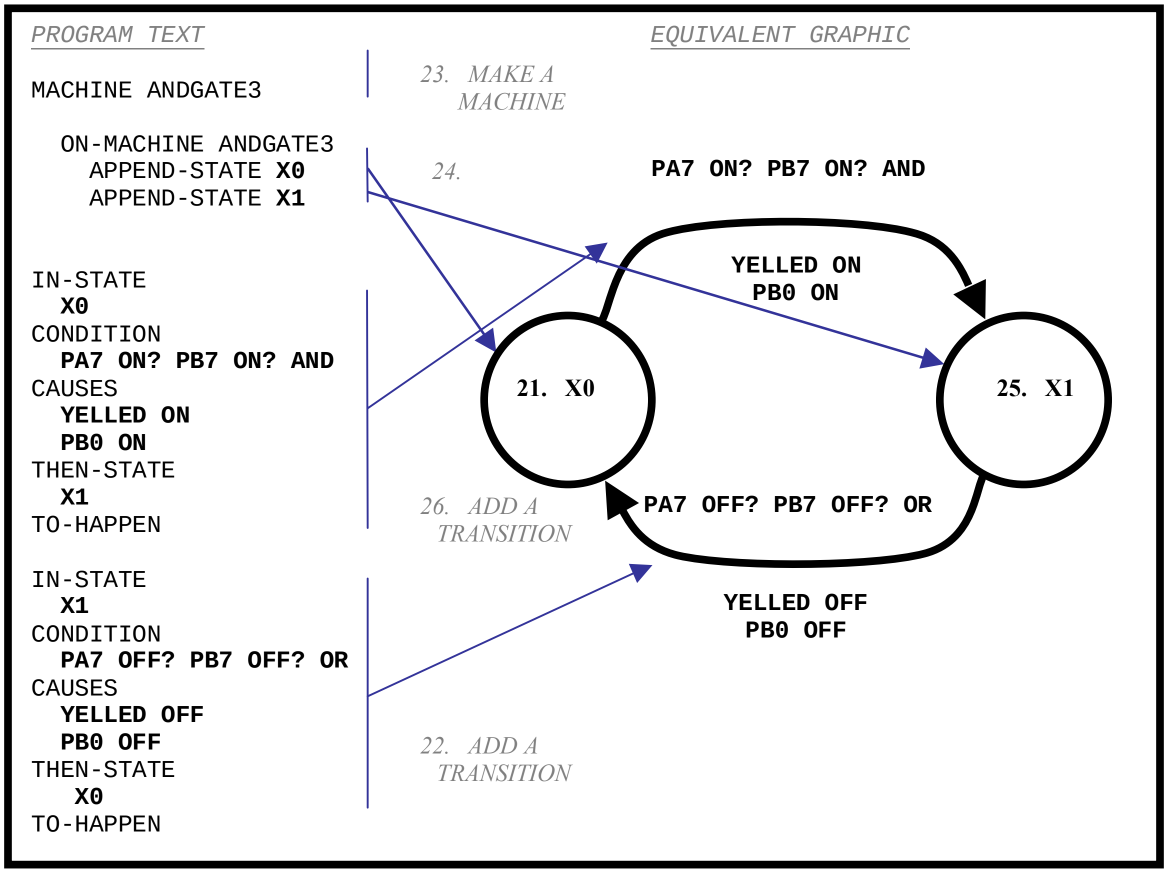
This, however, is another trick you should avoid. Again, why? This code does one of two actions every time the scheduler runs. The actions take longer than the Boolean test and transfer to another thread. The system will run slower, because the same outputs are being generated time after time, whether 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.

21. 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
```



```
PROGRAM TEXT

MACHINE ANDGATE3

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


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

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

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

22. 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 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.

23. 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.

24. BOUNCELESS+

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

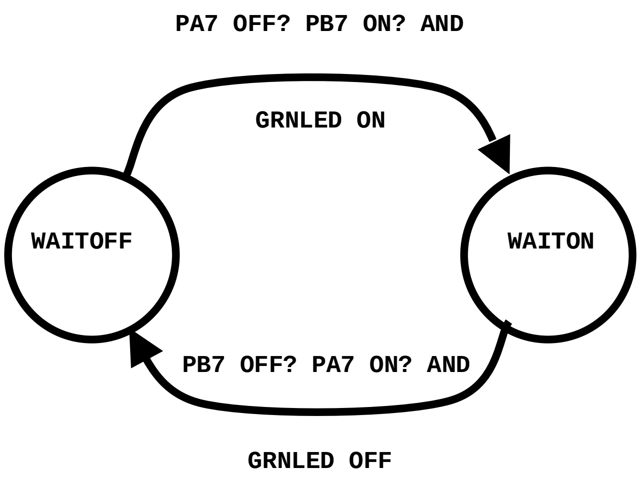
```
                STATE Y
IN-STATE          IN-STATE
  Y                 Y
CONDITION         CONDITION
  PA7 OFF?          PB6 OFF?
CAUSES            CAUSES
  GRNLED OFF        GRNLED ON
THEN-STATE        THEN-STATE
  Y                 Y
TO-HAPPEN         TO-HAPPEN
```

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

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	IN-STATE
WAITOFF	**WAITON**
CONDITION	CONDITION

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

25. 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	GOSUB DELAY	{ delay(x);	DELAY
BNE LOOP2			
LDAA #1	LET PB=TRUE	out(1,portA1);	LED-ON
STAA PORTA			
LDX # 0			
LOOP3 DEX	GOSUB DELAY	delay(x);	DELAY
BNE LOOP3			
LDAA #N	Let PB=FALSE	out(0,portA1);	LED-OFF
STAA PORTA			
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.

26. 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.

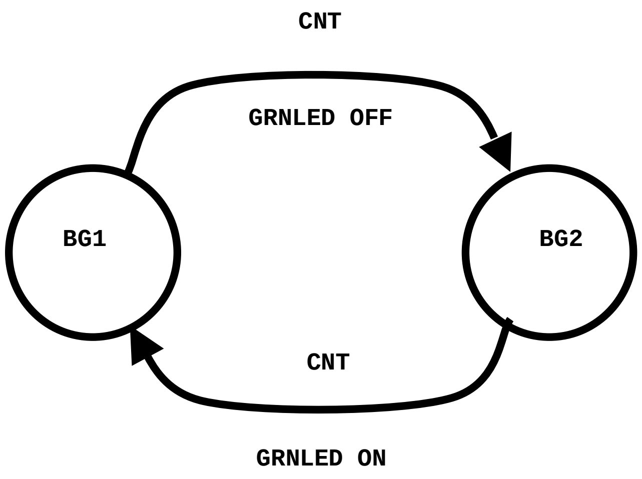
Since we're going to count, we need two variables to work with. One contains the count, the other the initial value we count down from. Let's add a place for those variables now, and initialize them

```
: -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



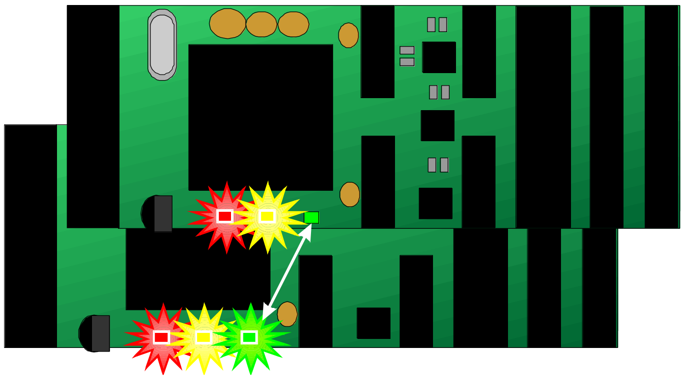
Above, the two transitions are "pretty printed" to make the four components of a transition stand out. As discussed previously, as long as the structure is in this order it could just as well been run together on a single line (or so) per transition, like this

```
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 reinitialized 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.

27. SPEED

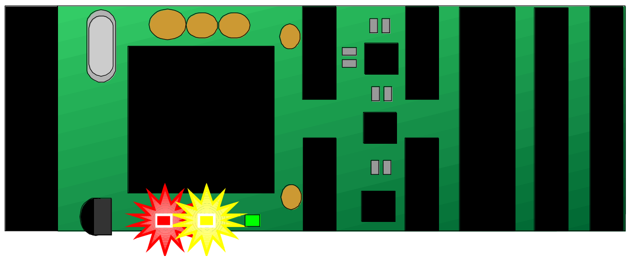
You've seen how to write a machine that delays based on a counter. Let's now try a slightly less useful machine just to illustrate how fast the IsoPod™ can change state. First reset your machine to get rid of the existing machines.

28. 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
```

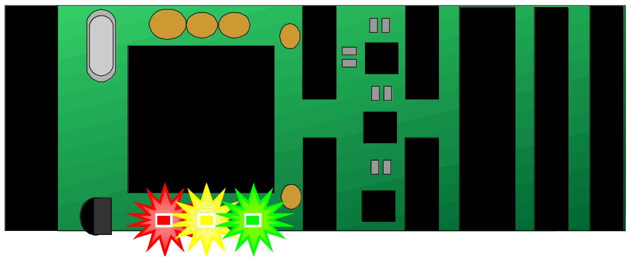
Now rather than install our new machine we're going to test it by running it "by hand" interactively. Type in:

```
ZIPON 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.

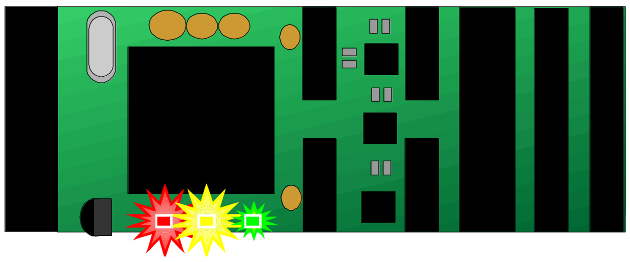
Now with the machine program and tested, we're ready to install the machine into the machine chain. The phrase to install a machine is :

```
    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.

29. 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:

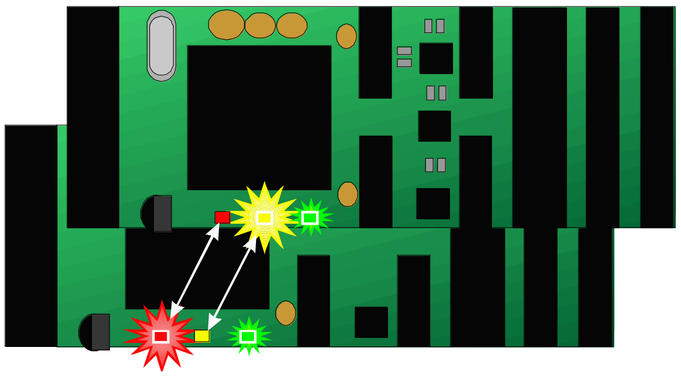
```
MACHINE REDYEL

  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
```

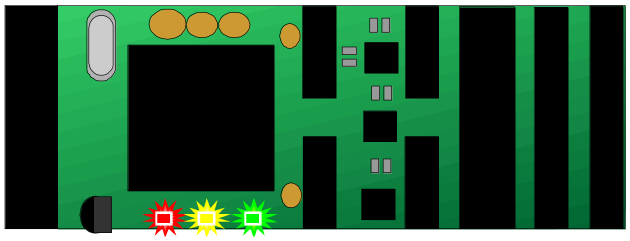
See the red and yellow LED's trade back and forth from on to off and vice versa.



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.

30. INPUT/OUTPUT 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.

31. Input 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) `TEST-MASK`, 2) `DATA-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
```

32. Set/Clear Output Trinaries

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> 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
```

33. And/Xor Output Trinaries

Sometimes, instead of setting and clearing bits, it’s more useful to replace or toggle bits. In this case, using the trinary operation mode of AND and XOR would be convenient. This mode requires: 1) a mask which is to be ANDed with the bits in the output port, 2) a mask which is to be XORed with the bits in the output port, and 3) the address of the I/O port. The keywords which proceed the parameters are, in order: 1) `AND-MASK`, 2) `XOR-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
```

The AND mask is applied *before* the XOR mask. This gives you four possibilities:
1) If a bit in the AND mask is 1, and the corresponding XOR bit is 1, that bit will be *toggled* (inverted) in the output port.

2) If a bit in the AND mask is 1, and the corresponding XOR bit is 0, that bit will be left unchanged in the output port.
3) If a bit in the AND mask is 0, and the corresponding XOR bit is 1, that bit will be *set* in the output port (cleared by the AND mask, and then inverted by the XOR mask).
4) If a bit in the AND mask is 0, and the corresponding XOR bit is 0, that bit will be *cleared* in the output port.

This is mainly useful when you want to toggle certain bits. But you can also use this to give the same function as SET-MASK and CLR-MASK:

```
HEX
DEFINE HEATER-ON  AND-MASK 0FD XOR-MASK 02 AT-ADDRESS 0FB0 FOR-OUTPUT
DEFINE HEATER-OFF AND-MASK 0FD XOR-MASK 00 AT-ADDRESS 0FB0 FOR-OUTPUT
DECIMAL
```

Note that 0FD is used for the AND mask. This is so that all of the bits other than PA1 remain unaffected. If the AND mask was set to 02, all the bits except PA1 would be cleared.

The value 0FD will also clear the PA1 bit. This is fine for HEATER-OFF, but for HEATER-ON we need to set that bit. So an XOR mask of 02 is applied to toggle the bit after clearing it, thus setting the bit.

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

34. *FLASH AND AUTOSTARTING*

35. Moving Your Application to Flash ROM

Up to now, all of the programs you have been downloading to the IsoPod™ have been stored in RAM memory. This is easy and quick for testing, and if your program gets stuck you can just reset the IsoPod™ or cycle its power, and you're back to the command processor. But eventually, you're going to want to install your application in ROM. Maybe you've run out of RAM space for your program: moving the tested part to ROM will free RAM for other purposes. Or maybe you want to make your application program permanent, so that it's ready to run when you turn on the IsoPod™.

The IsoPod™ has about 10K words (20K bytes) of Flash ROM available to hold your application. No special tools are required for this: you just need to add commands to your program file to tell IsoMax to put your program into ROM. These commands are `SCRUB`, `EEWORD`, and `IN-EE`, and are used at slightly different times, as we'll see in a moment.

1. Before you load your application, you should issue the `SCRUB` command. This will erase the "application" area in the Flash ROM, and return it to its factory-new state. This also erases any autostart information that may have been installed (we'll talk about this more below).

2. If you are using procedural code, each MaxForth word should be followed by the command `EEWORD` in the program file. 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`. In other words, you must say `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`. This is the one time where you don't use `EEWORD`. State transitions must use `IN-EE`.
   d. `MACHINE-CHAIN ... END-MACHINE-CHAIN` must be followed by `EEWORD`.
   e. `ON-MACHINE <name>` is ***not*** followed by any EE command.
   f. Trinaries, such as `DEFINE <name> ... FOR-INPUT` and `DEFINE <name> ... FOR-OUTPUT`, must be followed by `EEWORD`. `EEWORD` is placed after the end of the trinary definition.

When you use these words, the definitions you create are compiled into RAM and then moved into ROM. So, every time you use `EEWORD` or `IN-EE`, you free the RAM space that the latest definition was using. This keeps the maximum of RAM available.

Important: *do not intermix RAM and ROM definitions.* If you compile some definitions into RAM, and then move some others into ROM, and then compile some more into RAM, the program will run. But if you then reset the IsoPod™ or cycle its power, part of your program will be missing! What's worse, if you try to use the part that is in ROM, it will probably try to use some of the missing code from RAM, and crash the IsoPod™. The best approach is to test the first part of the program -- that is, the part that gets downloaded first -- and when it's working, move it all to ROM. Then download and test the next part of the program, move it to ROM, and so on.

36. Saving the RAM Variables

If, after you move your application into Flash ROM, you cycle the power of the IsoPod™, you'll see a strange result: your application will appear to be missing! This is because the IsoPod™ keeps some important information about your application in RAM variables. IsoMax also keeps some state machine information in RAM variables. When you cycle the power off and on, this information is lost.

Fortunately, there's an easy way to preserve this information. After you've finished loading your application program into ROM, simply type the command

```
    SAVE-RAM
```

This will preserve a copy of all the important variables in the Data Flash ROM. The next time you power up the IsoPod™, it will get this saved information and put it into RAM. As a bonus, any of *your* variables will be saved also. You can use this as a convenient way to initialize your application's variables and RAM data structures.

Remember: you *must* use `SAVE-RAM` after loading your application into Flash ROM.

37. Setting Your Program to Autostart on Reset

If you've followed the steps so far, your application program is permanently installed in Flash ROM, and will be visible when you first turn the IsoPod™ on. You can see this by cycling power to the IsoPod™, and then typing the `WORDS` command. But your program isn't running at this point -- only the IsoMax command interpreter is running. You need to start your program manually.

To start your program automatically, you need to install an **autostart vector.** This tells IsoMax what program to execute when the IsoPod™ is reset -- either when it's first powered on, or when a hardware reset occurs. IsoMax looks for possible autostart vectors only at certain addresses. These addresses are every 1K boundary in Program memory, that is, every multiple of $400 in the address space.

The autostart vector must be written at a location which
a) is a multiple of $400,
b) is in Flash ROM ($0000-$7DFF),
c) does not overlap the IsoMax kernel, and
d) does not overlap your application.

The suitable addresses (in hex) are:

IsoMax v0.5 and earlier	IsoMax v0.6 and later
5000	1800
5400	1C00
5800	2000
5C00	2400
6000	2800
6400	2C00
6800	3000
6C00	3400
7000	3800
7400	3C00
7800	
7C00	

Your application program will be loaded starting at the bottom of the available Flash ROM and working upward. So, the safest address to use for an autostart vector is near the *end* of the list. For IsoMax v0.5 and earlier, we recommend $7C00. For IsoMax v0.6 and later, we recommend $3C00.

Don't worry if your program overwrites some of the earlier autostart vector locations. If the location doesn't contain a special autostart pattern, IsoMax will treat it as ordinary program code and not an autostart vector. You don't need to "skip" the unused autostart locations.

Once you've chosen a location for your autostart vector, you need to tell IsoMax what word to execute on a reset. You do this with the `AUTOSTART` command. Let's suppose that you are want to start a word called `MAIN` and you want to put the autostart vector at $7C00. The command is:

```
      HEX 7C00 AUTOSTART MAIN
```

(Note the use of the word `HEX` to let you type a hex number.)

If you've done all the previous steps correctly, the IsoPod™ will now go directly to your application program (`MAIN`) when you reset or cycle the power.

38. Debugging Autostart Problems

If your autostart application doesn't work, you need to see what the IsoPod™ displays on the terminal. If you've disconnected your PC, re-connect its serial cable and run the terminal program (as described in Section 2, Getting Started). Then reset the IsoPod™ and see what is displayed.

a) If the "IsoMax" prompt is displayed and the command interpreter echoes characters that you type:

See if the IsoPod™ will accept a command. Try typing `WORDS` -- does it work? Do you see your application (e.g., `MAIN`) in the list of words?

a.1) If commands aren't recognized, you probably forgot to put all of your application into Flash ROM. (I.e., you forgot an `EEWORD` or `IN-EE` command somewhere.) You'll need to bypass the autostart (see below) and `SCRUB` your IsoPod™, then start over.

a.2) If commands are recognized, but your application doesn't appear in the list of words, you probably forgot to do a `SAVE-RAM`. You can't do `SAVE-RAM` now. You'll have to `SCRUB` the IsoPod and start over by reloading your application.

a.3) If your application does appear in the list of words, you probably forgot to install an autostart vector, or you installed it at a bad location. A quick test is to examine the autostart location with `PDUMP`. For example (with a vector at $7C00):

```
        HEX 7C00 8 PDUMP
```

The first location dumped should contain $A55A. If it contains $FFFF, you didn't install the autostart vector. If it contains any other value, this probably means that your program is using this memory, and you've tried to install an autostart vector that overlaps the program. You'll need to `SCRUB`, reload, and use a different autostart location.

a.4) If your application appears in the list of words, and your autostart vector appears in the dump, it's possible that your application is terminating. To test this, try typing your startup word as a command from the interpreter. For example,

```
        MAIN
```

If your program displays the "OK" prompt, that means it has finished running and has returned control to the command interpreter. This is a problem in your program.

If your program runs correctly from the command interpreter, and doesn't terminate, yet won't run from the autostart, it's time to contact the factory for help.

b) If the IsoPod locks up, and either does not respond to commands or displays continuous nonsense:

This could indicate a number of problems, including:

b.1) You didn't put all of your application into ROM.
b.2) You forgot to do `SAVE-RAM`, or you did it too soon (you must do `SAVE-RAM` *last,* after all of your application is compiled).
b.3) You installed the wrong word as the autostart vector, or your application program overlaps the autostart vector.
b.4) There's a problem with your application program.

The most common problem with application programs is forgetting to initialize something. With the IsoMax command interpreter, it's easy to issue commands which initialize an interface or a data structure. *Unless these commands are part of your program, their effect will be lost when the IsoPod™ is reset.* It's best to always explicitly initialize everything your program will use, at the beginning of your program.

> An easy way to test for this is to compile your program into ROM, do `SAVE-RAM`, but do *not* install the autostart vector. Cycle the power on the IsoPod™, making sure that you switch off long enough for RAM to be lost (say, 15 seconds or more). You should now be in the IsoMax command interpreter. Try to start your program by typing its name (e.g., `MAIN`). If it doesn't run, your initialization or your program logic may be at fault, but you can reset the IsoPod™ and use the command interpreter to explore the problem. We recommend that you always test your program this way before committing it to autostart.

Your program might also fail because of a design or logic error.

If your program runs correctly from the command interpreter after you've cycled the power, but won't autostart, call the factory for help.

Recovering from a locked IsoPod

Regardless of why the autostart failed, you need to restore the IsoMax command interpreter before you can fix it. To do this you must bypass the autostart. You do this by turning off the IsoPod™, installing a jumper from GND to PE4/SCLK, and then turning the IsoPod™ back on. (On the V2 IsoPod™, these are adjacent pins 2 and 4 on J5, and can be connected with a jumper block. On the V1 IsoPod™, these are adjacent pins 2 and 4 on J3.)

When you turn the IsoPod™ back on, you will be in the IsoMax command interpreter. Your application will still be in Flash ROM, but will not appear in the `WORDS` list. If you're an advanced programmer you may be able to glean some information by dumping memory, but for most of us the best thing to do is an immediate `SCRUB` command. `SCRUB` will erase your application, erase any autostart vectors, and restore the IsoPod™ to its factory default condition.

If this procedure does *not* restore the IsoMax command interpreter, your application has probably corrupted part of the IsoMax kernel. (This is very difficult, but not impossible, to do.) Contact the factory to have a new kernel programmed into your IsoPod™.

39. ISOMAX SYNTAX

40. State Machines Definition

```
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
```

The AT-ADDRESS/AS-TAG option is intended for state machine debugging, and is not used in normal applications.

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

The *boolean computation* must take nothing from the stack, and leave a true-or-false value on the stack. The *compound action* must take nothing from and leave nothing on the stack. If there is no action, you must still include the word CAUSES.

Instead of `TO-HAPPEN`, you can also use `NEXT-TIME` (which is identical). You can also use `THIS-TIME` to force the new state to be performed immediately.

The `IN-EE` command is optional. It will cause the condition phrase to be copied to Flash ROM. This must be used when compiling the state machine to ROM.

41. State Machines Installation

```
<name-of-state> SET-STATE
```

`SET-STATE` makes the given state current in its parent state machine. This *must* be done before the machine is activated with `INSTALL` or `SCHEDULE-RUNS`. (In a machine chain, all the individual machines must have `SET-STATE` before the chain is activated.)

```
MACHINE-CHAIN <name-of-chain>
     <name-of-machine>
     ...
     <name-of-machine>
END-MACHINE-CHAIN
```

```
INSTALL <name-of-machine>
INSTALL <name-of-chain>
```

`INSTALL` is the preferred method to activate state machines. You are limited to 16 `INSTALL`s. If you need to run more than 16 state machines, create a machine chain (which can be any length) and then `INSTALL` the chain.

```
UNINSTALL
```

Removes the last INSTALLed machine. May be used repeatedly to remove previously isntalled machines, last in, first out.

`NO-MACHINES`

Removes all INSTALLed machines.

42. State Machines Timing and Control

`EVERY <n> CYCLES SCHEDULE-RUNS <name-of-chain>`

This is the original method used in early IsoMax kernels to activate state machines. It is retained for backward compatibility, but `INSTALL` is preferred. If `SCHEDULE-RUNS` is used, it will override all `INSTALL`ed machines. <n> specifies the IsoMax processing cycle, and is in counts of a 5 MHz clock.

`<n> PERIOD`

Changes the IsoMax processing cycle. <n> is in counts of a 5 MHz clock.

`<name-of-state> IS-STATE?`

Returns true if the given state is the current state in its parent state machine.

`ISOMAX-START`

Starts IsoMax processing. The machine list is cleared (no machines are installed). This also cancels the action of any previous `SCHEDULE-RUNS` command.

`STOP-TIMER`

Halts IsoMax processing. **Note also that COLD and SCRUB will halt IsoMax, clearing the machine list and cancelling any previous SCHEDULE-RUNS command.**

43. Input/Output Trinaries

```
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> AND-MASK <n> XOR-MASK <n> AT-ADDRESS <a>
FOR-OUTPUT
```

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

44. Performance Monitoring Variables

The following are variables which can be examined, and in some cases stored, by the user.

`TCFTICKS` is incremented once per IsoMax processing cycle. If the previous state machine processing had not completed -- an "overrun" condition -- `TCFTICKS` will be incremented but the state machine will not be restarted.

`TCFOVFLO` contains a count of the number of overruns that have occurred. An "overrun" occurs when the IsoMax service has not finished by the time of the next clock interrupt. This indicates that the `CYCLES` parameter is too small.

`TCFALARM` contains an "alarm limit" for `TCFOVFLO`. If zero (the default state), overruns are counted but otherwise ignored. If nonzero, when this many overruns have occurred, the MaxForth word indicated by `TCFALARMVECTOR` is executed.

`TCFALARMVECTOR` contains the CFA of a MaxForth word which will be executed when the overrun alarm limit is reached. It is the user's responsibility to reset the `TCFOVFLO` counter, if desired. Note that this routine will be executed as an interrupt, and on return, the IsoMax service routine will continue. If this variable contains zero (the default), no service will be performed.

`TCFOVFLO`, `TCFALARM`, and `TCFALARMVECTOR` are reset to zero by `SCHEDULE-RUNS` and `ISOMAX-START`.

The following variables measure CPU utilization. Each time the clock interrupt is serviced, the scheduler measures the duration of the service, i.e., the number of timer cycles used to process the state machine. Currently the timer counts at 5 MHz, so 5 cycles = 1 usec.

`TCFMAX` contains the maximum observed duration.
`TCFMIN` contains the minimum observed duration.
`TCFAVG` contains a running average duration.

These three variables are reset to zero by `SCHEDULE-RUNS` and `ISOMAX-START`.

45. I/O PROGRAMMING

OK, so now you know how to make and run state machines using IsoMax. Your state machines need to do something! This is where the rich assortment of inputs and outputs on the IsoPod™ comes into play. You can use turn pins on and off, check logic levels, send pulse streams, measure time, read analog voltages, send and receive serial data, and control SPI (Serial Peripheral Interface) chips.

All of the input/output functions of the IsoPod™ follow a simple pattern: you specify the IsoPod™ pin, and then the action you want to perform. If you've encountered object-oriented programming before, this will be familiar to you. You specify an *object* (an I/O pin), and then you perform a *method* (an input or output action).

Syntax note: an object and method are always a *pair*. Normally, they must appear together in your program. We'll explore some ways later to get around this limitation, but for now, remember that you must always specify both.

46. Bit Output

You've already seen examples of the simplest kind of I/O: turning an output pin on or off. We used this in Section 4 to turn LEDs on and off. The basic commands are

```
pin ON
pin OFF
```

where "pin" can be any one of the following:

```
REDLED GRNLED YELLED
PA0 PA1 PA2 PA3 PA4 PA5 PA6 PA7
PB0 PB1 PB2 PB3 PB4 PB5 PB6 PB7
PD0 PD1 PD2 PD3 PD4 PD5
PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7
TA0 TA1 TA2 TA3
TB0 TB1 TB2 TB3
TC0 TC1
TD0 TD1 TD2 TD3
PWMA0 PWMA1 PWMA2 PWMA3 PWMA4 PWMA5
PWMB0 PWMB1 PWMB2 PWMB3 PWMB4 PWMB5
```

That's a lot of outputs! But don't look for TA0-3 and TB0-3 on the connectors. These are dual-function pins, and on the connector are labeled differently:

TA0 = PHASEA0	TB0 = PHASEA1
TA1 = PHASEB0	TB1 = PHASEB1
TA2 = INDEX0	TB2 = INDEX1
TA3 = HOME0	TB3 = HOME1

Also, on the V2 IsoPod, `PD0`, `PD1`, `PD2` are the same as `REDLED`, `YELLED`, `GRNLED` (these are the pins that control the LEDs). Pins `PE0`, `PE1`, `PD6`, `PD7` are reserved for the SCI channels, and not available for simple I/O. There are no pins `TC2` and `TC3`.

So, you can turn the red LED on with
```
     REDLED ON
```
and turn it off with
```
     REDLED OFF
```

What if you want to set an output on or off, depending on the value of a variable? You could write an IF..ELSE..THEN using ON and OFF. But a simpler solution is:

`n  pin SET`	Sets the output of the pin according to "n". If n is zero, turns the pin off. If n is nonzero, turns the pin on. (Zero and nonzero correspond to logical false and true.)

So,

`1 REDLED SET`	will turn the LED on,
`0 REDLED SET`	will turn the LED off, and
`33 REDLED SET`	(or any nonzero number) will turn the LED back on.

Perhaps you want to flip the state of the pin, but you don't know whether it was previously turned on or off:

`pin TOGGLE`	will change the state of the pin. If it was on, this turns it off. If it was off, this turns it on.

If you need to know whether a pin has been previously turned on or off, you can ask with:

`pin ?ON`	returns true if the pin has been turned on.
`pin ?OFF`	returns true if the pin has been turned off.

We know, these are redundant, since `?OFF` is the logical inverse of `?ON`. We give you both of them so you can use whatever makes your program most readable.

47. Bit Input

Many of the programmable output pins can be used instead as logic input pins:
```
     PA0 PA1 PA2 PA3 PA4 PA5 PA6 PA7
     PB0 PB1 PB2 PB3 PB4 PB5 PB6 PB7
     PD0 PD1 PD2 PD3 PD4 PD5
     PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7
     TA0 TA1 TA2 TA3
     TB0 TB1 TB2 TB3
     TC0 TC1
     TD0 TD1 TD2 TD3
```

Obviously the LEDs can't be used as digital inputs. And the PWM pins on the IsoPod™ are permanently configured as output pins. The rest of the `Pxx` and `Txx` pins can be used as inputs or outputs, under program control.

There are 14 new pins that can *only* be used as digital inputs:

```
ISA0 ISA1 ISA2
     FAULTA0 FAULTA1 FAULTA2 FAULTA3
     ISB0 ISB1 ISB2
     FALUTB0 FAULTB1 FAULTB2 FAULTB3
```

To read a digital input pin, you can use the commands

`pin ON?`	returns true if the pin is at a logic high.
`pin OFF?`	returns true if the pin is at a logic low.

Again, these are just two different ways of looking at the same input. Use whatever makes your program more readable.

Note that these are not the same as the `?ON` *and* `?OFF` *functions shown above.* There are two important differences:

a) `?ON ?OFF` return the last value that was written to the pin. If the pin has been configured as an input, or as an open-collector output, this may not be the actual logic level! `ON? OFF?` return the actual logic level on the pin.

b) `ON? OFF?` will change the pin from an output to an input. `?ON ?OFF` will not change the pin's configuration; if it was an output, it remains an output.

The rule is this: use `?ON ?OFF` for output pins. Use `ON? OFF?` for input pins.

We haven't talked about how to configure a pin as a digital output or a digital input. That's because you don't have to – it's automatic. If you use one of the output words like `ON` or `TOGGLE`, IsoMax will automatically configure that pin as an output (if it hadn't already done so). Likewise, if you use `ON?` or `OFF?`, IsoMax will automatically configure that pin as an input. (You can even switch a pin from output to input, or input to output, in your program…but that's an unusual application.)

48. Byte Input and Output

Port A and port B on the IsoPod™ are 8-bit parallel I/O ports that are entirely available for you to use. You can use the individual pins of these ports for single-bit input and output, as we've just described. (The pin names are PA0-PA7 for port A, and PB0-PB7 for port B.) Or, you can use either or both of these ports as 8-bit parallel ports.

To tell IsoMax that you want to treat all 8 pins as a single byte, you use the port names:

```
PORTA     PORTB
```

On port A, PA0 is the least-significant bit, and PA7 is the most-significant bit. Likewise for port B.

There are only two actions that you can perform on an 8-bit parallel port:

`port GETBYTE`	reads the 8-bit value from the (input) port
`port PUTBYTE`	writes an 8-bit value to the (output) port

Again, the configuration is automatic. When you use `GETBYTE`, *all* of the pins of the port are configured as inputs. When you use `PUTBYTE`, all eight pins are configured as outputs.

To turn all of the port A bits off, except PA7 which is turned on, you could use:

```
HEX 80 PORTA PUTBYTE
```

To test whether any of the low 4 bits of port B are on, you could use

```
PORTB GETBYTE  HEX 0F AND
```

which will return a nonzero value if any of the bits PB0-PB3 are high. Here's a trivial example of a program that makes the IsoPod™ into an eight-bit inverter:

```
PORTA GETBYTE  INVERT  PORTB PUTBYTE
```

In this example, a byte is read from port A. It is then logically inverted, and written to port B. (Of course, this will only happen once. To respond to changes in the port A inputs, this bit of code would have to be written in a loop, or into an IsoMax state machine, so that it is called repeatedly.)

49. Serial Communications Interface (SCI)

The IsoPod™ includes *two* full-duplex asynchronous serial ports. These are named

```
SCI0      SCI1
```

Note that you do *not* refer to the serial ports by their pin names, but by their *port* names. `SCI0` is the RS-232 port that is connected to your PC for software development (pins SOUT and SIN on connector J1). `SCI1` is the RS-232/RS-422 port, pins SOUT1 and SIN1 on connector J4 (on the V1 IsoPod™, only RS-422 is available for this port).

The basic operations on the serial port are `TX` and `RX`:

`port TX`	transmit one byte on the serial output
`port RX`	receive one byte on the serial input

For example, to send the character "A" (hex 41) to the terminal (connected to the primary RS-232 port), you could use the command

```
HEX 41  SCI0 TX
```

To receive a character from the RS-422 port, and display its hex value, you could use

```
SCI1 RX  HEX .
```

But before you use `SCI1`, you must set its baud rate…

50. Setting the Baud Rate

When the IsoPod™ is reset, it sets the SCI0 port to operate at 9600 baud. You can change this to some other value, say 38400, with the command

```
DECIMAL 38400  SCI0 BAUD
```

(Baud rates are normally written as decimal numbers.) The moment you execute this command, the baud rate will take effect, so you won't see the usual "OK" response. You'll have to change the baud rate of MaxTerm or HyperTerminal (or whatever you are using) to the new rate. Then you can press Enter and see the response at the new rate.

Before you use the SCI1 port, you *must* set its baud rate. For example,

```
DECIMAL 9600  SCI1 BAUD
```

For the baud rate, you can specify any value between 300 and 57600. The "standard" baud rates 300, 600, 1200, 2400, 4800, 9600, 19200, and 38400 will be accurately set. For other values, the IsoPod™ will give the best approximation that it can, within the limits of its baud rate generator.

(The baud rate is produced by dividing 2.5 MHz by an integer. Thus 9600 baud is produced by dividing 2.5 MHz by 260, which gives an actual rate of 9615.4 baud, close enough for serial communications. But the closest we can come to 57600 baud is dividing by 43 to get 58140 baud.)

51. Polling the SCI Status

`RX` will *wait* for a character to be received, unless there's already one waiting in the serial data register. This may lead to "Program Counter Capture," where the processor sits in a loop waiting forever for an external event. You want to avoid this when you write IsoMax programs!

The solution is to *poll* the SCI receiver. You do this with

```
        port RX?    check to see if a receive character is available
```

RX? will never wait. It will instantly return a true (non-zero) value if a character is available, or a zero value if no character is waiting in the receiver. It does *not* fetch the character from the receiver. If RX? returns true, you must follow it with RX to get the character.

We'll see an example of how to use this soon.

TX might also wait, if a previous character hadn't finished transmitting. But at least this wait won't be indefinite: you know that the transmitter will send the character in a short time, and so you'll have to wait at most one character period. But this might also be a problem in IsoMax code, so you can check to see if the transmitter can accept a character with

```
        port TX?    check to see if transmitter is ready for a character
```

TX? will instantly return a true (non-zero) value if the transmitter can accept a character *now*. It will return a zero value if the transmitter is busy, that is, if the transmitter is still sending the last character and can't accept a new one yet.

52. Serial Receive Buffering

What happens if receive characters arrive faster than you're checking for them? With most serial ports, if a second character arrives before you've read the first one, you get an *overrun* condition and one of the two characters is lost. This is a problem!

Fortunately, the IsoPod™ has a built-in solution for this problem. If you wish, you can define a *receive buffer* which will hold characters until you're ready to process them. This buffer can be as big as you like (limited of course by the amount of available RAM).

To activate receive buffering, you must first reserve some RAM for the buffer. An easy way to do this is to define an IsoMax variable, and then immediately allocate some extra RAM for it. To reserve a buffer of 20 (decimal) characters, you could type

```
        VARIABLE BUFFER1  DECIMAL 19 ALLOT
```

The reason we allot 19 instead of 20 characters is because VARIABLE will allocate room for one character. So, one character in the VARIABLE plus an extra allotment of 19 will give 20 characters of storage.

However, you should be aware that this buffer will actually hold only 16 serial characters. The reason is that 4 characters' worth of storage will be used for control information. So, when you are sizing your buffer, remember to add 4 for this "overhead." IsoMax won't let you use a buffer size smaller than 5.

Next you tell the IsoPod™ where that buffer is located, how big it is, and what port to use it for.

```
    BUFFER1 20  SCI1 RXBUFFER
```

This says that `BUFFER1`, with a length of 20, is to be used as the receive buffer for port `SCI1`. (Note that you use the real buffer length, 20, and not 19.)

That's all there is to it! The buffer is now active and will begin storing received characters. None of your other serial code has to change: `RX?` will tell you if there's a character waiting in the buffer, and `RX` will fetch the next character from the buffer.

53. Serial Transmit Buffering

Transmitted characters will never get lost (at least not by the IsoPod™), because the IsoPod™ will always wait until it can send a character. But in a Virtually Parallel application, that wait might prevent other tasks from being accomplished. For example, if a particular state in a state machine needs to send a 16-character message at 9600 baud, that can add a 16.7 millisecond delay – very noticeable when IsoMax is running state machines every 10 milliseconds!

Again, there is a built-in solution. You you can define a *transmit buffer* which will hold a block of characters and then dole them out automatically to the SCI transmitter. Again, your buffer size is limited only by RAM.

Transmit buffering is activated the same as receive buffering. First reserve some RAM for the buffer. Of course, we can't use the same buffer at the same time for transmitting and receiving, so we'll define a new buffer for transmitting:

```
    VARIABLE BUFFER2  DECIMAL 19 ALLOT
```

Next tell the IsoPod™ where that buffer is located, how big it is, and what port to use it for.

```
    BUFFER2 20  SCI1 TXBUFFER
```

This is just like the previous example except that we're using `BUFFER2`, and we're using it as a `TXBUFFER` (transmit buffer).

That's it! The buffer is now active and will store characters that you transmit. None of your other serial code has to change: TX? will tell you if there's room in the buffer, and TX will transmit a character via the buffer.

54. Terminal I/O

IsoMax uses serial port SCI0 for its to connect to a serial terminal (or a terminal program such as MaxTerm or HyperTerminal). The "customary" terminal input and output operations still work in IsoMax, as follows:

KEY performs the same function as SCI0 RX
EMIT performs the same function as SCI0 TX
?TERMINAL performs the same function as SCI0 RX?
?KEY performs the same function as SCI0 RX?

(?TERMINAL and ?KEY are equivalent. ?TERMINAL is the older name for this function, and is retained for backward compatibility.) You can freely intermix KEY and SCI0 RX, or EMIT and SCI0 TX, with no confusion.

This also means that you can change the baud rate of the terminal with SCI0 BAUD. And, if you specify a receive buffer with SCI0 RXBUFFER, that also will be used for terminal input. This is especially useful when downloading files to the IsoPod™.

55. A Serial I/O IsoMax Example

Here's how you might use RX? in a state machine. This machine will listen on serial port SCI1. When it sees an ASCII "1" character (hex 31), it will turn on the red LED. An ASCII "0" (hex 30) will turn off the red LED. All other characters are ignored.

At first you might be tempted to write the state machine this way:

```
HEX
MACHINE WATCHSCI1
  ON-MACHINE WATCHSCI1
    APPEND-STATE WAITCHAR
    APPEND-STATE TESTCHAR

IN-STATE WAITCHAR  CONDITION SCI1 RX?  CAUSES ( no action ) THEN-STATE
  TESTCHAR TO-HAPPEN

IN-STATE TESTCHAR  CONDITION SCI1 RX 30 = CAUSES  REDLED OFF  THEN-STATE
  WAITCHAR TO-HAPPEN

IN-STATE TESTCHAR  CONDITION SCI1 RX 31 = CAUSES  REDLED ON  THEN-STATE
  WAITCHAR TO-HAPPEN

WAITCHAR SET-STATE   INSTALL WATCHSCI1
```

The first state, WAITCHAR, is fine. The machine will stay in this state until a character is received. But TESTCHAR won't work, because it tries to read the SCI1 port *twice*. (Once for each condition.) The first time it will get the character, but the second time it will try to read *another* character…and of course, there isn't a second character.

To solve this we need to use an auxiliary variable to hold the character. Then we can read it only once, and test it several times.

```
VARIABLE CMDCHAR
HEX
MACHINE WATCHSCI1
  ON-MACHINE WATCHSCI1
    APPEND-STATE WAITCHAR
    APPEND-STATE TESTCHAR

IN-STATE WAITCHAR  CONDITION SCI1 RX?  CAUSES  SCI1 RX CMDCHAR C! THEN-STATE
  TESTCHAR TO-HAPPEN

IN-STATE TESTCHAR  CONDITION CMDCHAR C@ 30 = CAUSES  REDLED OFF  THEN-STATE
  WAITCHAR TO-HAPPEN

IN-STATE TESTCHAR  CONDITION CMDCHAR C@ 31 = CAUSES  REDLED ON  THEN-STATE
  WAITCHAR TO-HAPPEN

WAITCHAR SET-STATE   INSTALL WATCHSCI1
```

There's one more possible problem with this machine. What if we get a character that's neither 30 nor 31? We'll see the character, and make the transition to TESTCHAR state. But since no condition is satisfied, we never leave TESTCHAR state! Thus we never return to WAITCHAR state and we never accept another character. This is a flaw in the design of our state machine; fortunately, it's easily fixed by adding another transition:

```
IN-STATE TESTCHAR  CONDITION CMDCHAR C@ 30 <  CMDCHAR C@ 31 > OR  CAUSES
  ( no action )  THEN-STATE WAITCHAR TO-HAPPEN
```

Now, if the character is neither 30 nor 31, the machine will perform no output, but it will return to wait for another character.

56. Serial Peripheral Interface

The IsoPod™ includes a Serial Peripheral Interface (SPI) for communication with peripheral chips and other microprocessors. For consistency with other usage, and to make provision for future expansion, the port is named

```
SPI0
```

The basic operations on the SPI port are TX and RX, TX? and RX?:

`port TX-SPI`	transmit one word on the SPI output

`port RX-SPI`	receive one word on the SPI input
`port TX-SPI?`	check to see if transmitter is ready for a word
`port RX-SPI?`	check to see if a received word is available

However, an SPI port does not work like a normal serial port. In the SPI port, the transmitter and receiver are linked. Whenever you transmit a word, you receive a word. Also, the behavior of the port depends on whether you are operating as an SPI Master or an SPI Slave:

Master – You start an SPI transaction by writing a word to the SPI transmitter (with `TX-SPI`). Every time you do this, a word will be loaded into the receive register. So, after every `TX-SPI`, you should do an `RX-SPI` to read this received word and make the register ready for a new word. (The receive register is loaded even if the slave device doesn't output a reply.)

Slave – You wait for data to be sent you to by the SPI Master. When this happens, `RX-SPI?` will return true, and you can get the word with `RX-SPI`. Any data that you want to send to the Master must be *preloaded* into the transmit register with `TX-SPI`, because it will be sent *as you are receiving* the word from the Master. Every time you receive a word, the transmitter will be emptied. If you don't load a new word into the transmitter, it will keep sending the last word you loaded.

More differences are that the word size can range from 2 to 16 bits, and can be sent LSB-first or MSB-first.

57. Setting the SPI Parameters

The Master and Slaves must agree on the SPI data format and rate. These options are controlled by the following commands:

`n port MBAUD`	Sets the baud rate to "n" Mbaud, where n is 1, 2, 5, or 20. (The actual rates are 1.25, 2.5, 5, or 20 Mbaud, but the `MBAUD` command expects an integer value.) The baud rate only needs to be set on the Master; this will automatically control the Slaves.
`n port BITS`	Specifies the number of bits "n" to be sent by `TX-SPI` and read by `RX-SPI`. n may be 2 to 16.
`port MSB-FIRST`	Specifies that words are to be sent and received most-significant-bit first.
`port LSB-FIRST`	Specifies that words are to be sent and received least-significant-bit first.

Master and Slaves must also agree on *clock phase* and *clock polarity*. In the DSP56F80x processors these are controlled by the CPHA and CPOL bits in the SPI Control Register. In IsoMax they are controlled with these commands:

`port LEADING-EDGE`	Receive data is captured by master & slave on the first (leading) edge of the clock pulse. (CPHA=0)
`port TRAILING-EDGE`	Receive data is captured by master & slave on the second (trailing) edge of the clock pulse. (CPHA=1)
`port ACTIVE-HIGH`	Leading and Trailing edge refer to an active-high pulse. (CPOL=0).
`port ACTIVE-LOW`	Leading and Trailing edge refer to an active-low pulse. (CPOL=1).

Once the communication parameters have been set, the SPI port should be enabled as either a Master or a Slave:

`port MASTER`	Enables the port as an SPI Master. MOSI is output, MISO is input, and SS has no assigned function. (The SS pin may be used as GPIO output bit PE7.)
`port SLAVE`	Enables the port as an SPI Slave. MOSI is input, MISO is output, and SS is the Slave Select input. SS must be low for the SPI port to receive and transmit data.

Remember that the SPI port is not activated until you use `MASTER` or `SLAVE`.

58. Serial Receive Buffering (version 0.6)

Like the SCI ports, the SPI port may use a receive buffer. The format and requirements are exactly the same as for the SCI port: the buffer must be at least 5 cells long, and is installed with the command

```
address length port RXBUFFER
```

This is particularly valuable on SPI Slaves, since data can be sent to them at any time from the Master. If you don't have a receive buffer on the Slave, you'd have to check the receiver constantly for new data…because if the transmitter sent two words before you checked, you'd lose one. But even at 20 Mbaud, the buffered receiver won't lose data – unless of course you overflow the buffer! (The receive buffer uses *interrupts*, which means that the instant a full word has been received, the processor can store it in the buffer.)

Buffering is less important on an SPI Master, because the Master always has complete control over when data will be received. Data is received when data is sent! But if you're using a transmit buffer to send a block of SPI data without waiting, you should have a receive buffer at least as big, since every word send will cause a word to be received.

When the receive buffer is active, `RX-SPI` and `RX-SPI?` work exactly as before.

Specifying any address with a length of zero will disable the receive buffer and return to "unbuffered" operation. For this, you can even use an address of zero, e.g.,

```
0 0 SPI0 RXBUFFER
```

59. Serial Transmit Buffering (version 0.6)

The SPI port may also use a transmit buffer. Again, the buffer must be at least 5 cells long, and is installed with the command

```
address length port TXBUFFER
```

This is valuable for Slaves, because the Slave doesn't know when the Master will ask for a word of data. (When the Master sends a word, it expects the Slave to return one.) When the transmit buffer is active in a Slave, the first word sent will be preloaded into the SPI transmitter. Additional words will be held in the transmit buffer. Each time the Master transfers a word over the SPI port, the Slave's transmitter will be automatically loaded with the next word to be sent.

You should be aware that some SPI applications can't benefit from preloaded data in the buffer. Sometimes, the slave must receive a command word from the Master, and then generate a reply based on that command. In this case, we don't know what to load into the transmitter until the received word has been processed, so we can't "preload" a reply into the transmit buffer.

Many SPI applications involve this kind of exchange, so often there is no advantage to transmit buffering on the Master. The Master always has complete control over the data flow, so there's no danger of its transmitter running out of data. But if you are using the SPI transmitter to send a block of data, and you don't want stop other Virtually Parallel processing, you could load the entire block into a transmit buffer.

As before, specifying any address with a length of zero will disable the receive buffer and return to "unbuffered" operation. For example,

```
0 0 SPI0 TXBUFFER
```

60. An SPI Master-Slave Example

Here's a simple procedural program that configures an IsoPod™ as an SPI Slave device. It awaits a 16-bit value on the SPI port. When it receives a 16-bit value, it treats that value as an address, fetches that location in Program memory, and then returns that 16-bit value the next time the Slave receives a word.

```
DECIMAL
VARIABLE TBUF 16 ALLOT
VARIABLE RBUF 16 ALLOT

: SLAVE-MAIN
    16 SPI0 BITS  SPI0 MSB-FIRST  SPI0 TRAILING-EDGE
    SPI0 ACTIVE-LOW  SPI0 SLAVE
    TBUF 16 SPI0 TXBUFFER
    RBUF 16 SPI0 RXBUFFER

    \ simple SPI slave P-memory dump
    \ 0000 = null command, discarded, no reply
    \ nnnn = address.  On next xmit, send memory contents.
    BEGIN ?KEY 0= WHILE
      SPI0 RX-SPI? IF
        SPI0 RX-SPI ?DUP IF
            P@ SPI0 TX-SPI
        THEN
      THEN
    REPEAT ;
```

The outer loop of the program checks for a keypress on the RS-232 terminal input. If a key is detected, the slave program terminates. Otherwise, it checks to see if a word has been received on the SPI port with `SPI0 RX-SPI?` If a word has arrived, it is obtained with `RX-SPI`. If it is nonzero (tested with `?DUP`), it is used as the address for `P@` (fetch from Program memory), and the resulting data is sent to the transmitter with `TX-SPI`. The loop then continues.

Observe that we don't send anything in response to a 0000 command code. This primitive SPI protocol depends on the Master and Slave staying in perfect synchronization. Every word received generates one word of reply; and that reply word will be expected on the next transmission from the Master. Should the Slave ever get "ahead" or "behind" the Master – say, by losing a word -- it will stay ahead or behind, indefinitely. All but the very simplest SPI protocols must be designed to handle this problem, and recover automatically. In this example, the Master can send 0000 codes to read out the Slave's transmit buffer without refilling it with new data. (A more sophisticated protocol might require a very specific message format with "command" and "data" bytes, checksums, and so forth.)

Here's the companion program which runs on a second IsoPod™ as an SPI Master.

```
: SEND ( x -- x' )
    PE7 OFF  SPI0 TX-SPI  SPI0 RX-SPI   PE7 ON  ;
```

```
DECIMAL
: SLAVE@ ( a -- n )
    SEND DROP      ( send address, discard reply )
    250 0 DO LOOP  ( give slave time to respond )
    0 SEND         ( send null to fetch queued value )
;

: RDUMP ( a n -- )
    16 SPI0 BITS  SPI0 MSB-FIRST SPI0 TRAILING-EDGE
    SPI0 ACTIVE-LOW  1 SPI0 MBAUD  SPI0 MASTER
    \ Remote slave P-memory dump
    OVER + SWAP DO
        CR I 5 U.R  2 SPACES
        I 8 + I DO
            I SLAVE@ 5 U.R
        LOOP
    8 +LOOP
;
```

The key word in this program is `SEND`. Given a value on the stack, `SEND` will pull the Slave Select line low (active), transmit the value over the SPI port, receive the value which is returned from the slave, and then pull the Slave Select line high (inactive). This assumes that the SPI ports of the Master and Slave IsoPods are connected directly together, as follows:

Slave		Master
GND	↔	GND
PE4/SCLK	↔	PE4/SCLK
PE5/MOSI	↔	PE5/MOSI
PE6/MISO	↔	PE6/MISO
PE7/SS	↔	PE7/SS

Note that on the Slave, the PE7 pin is used as SS (Slave Select), and must be pulled low before the Slave will accept or send SPI data. But on the Master, PE7 is just a general-purpose output pin. What we're really doing is connecting the Slave's SS input to the Master's PE7 output….they just happen to use the same pin on the I/O connector.

Remember also that every time the Master sends a word over the SPI, it will receive a word back. This is handled by `SEND` which waits for the received word (with `RX-SPI`) after every transmission. This performs another subtle but important function: you can't pull Slave Select high until the SPI transmission is *finished.* `TX-SPI` won't wait for the 16 bits to be transmitted; it will return as soon as they're loaded into the transmit buffer. It will take over 12 microseconds (at 1.25 Mbaud) to send those bits! But `RX-SPI` won't have a result until 16 bits have been sent, and 16 bits received in reply. So waiting for `RX-SPI` ensures that the transmission is complete. For this application, it's best that the Master *not* use transmit and receive buffers.

`SEND` or something like it will probably be a key word in any SPI Master application. With it, we can construct `SLAVE@` ("slave fetch"). Given an address on the stack, `SLAVE@` sends that to the Slave, and discards whatever the slave sends back (the reply is meaningless, since the Slave doesn't have an address yet). Then the Master must wait for a short delay, because the Slave has to have time to see that it has received a command, process the command, and put the reply in its transmit buffer. Finally the Master sends a 0000 command code. The very action of sending this 0000 value will cause the data in the Slave's transmit buffer to be sent back to the Master. This is the reply we desire from the slave, so `SLAVE@` returns with this on the stack.

`RDUMP` ("remote dump") is a command very much like `DUMP`, but it uses `SLAVE@` to dump memory from the Slave via the SPI port. Given an address 'a' and length 'n', the outer DO loop steps through the addresses 8 at a time. The inner DO loop steps through each block of 8 addresses one at a time, fetches the data from the Slave, and prints that data.

Incidentally, note that we set the baud rate on the Master, but not on the Slave. The Slave always follows the Master's baud rate. But `MSB-FIRST`, `TRAILING-EDGE`, `and` `ACTIVE-LOW` must be set independently on Master and Slave (and they must match).

This program can be modified to return any kind of data from the Slave. For example, the Master could send an ADC channel number, and the Slave could read that channel and send the result.

61. PWM Output

The IsoPod™ can generate pulse-width-modulated (PWM) square waves on 25 different output pins. These pins are

```
TA0 TA1 TA2 TA3
TB0 TB1 TB2 TB3
TC0 TC1
TD0 TD1 TD2[1]
PWMA0 PWMA1 PWMA2 PWMA3 PWMA4 PWMA5
PWMB0 PWMB1 PWMB2 PWMB3 PWMB4 PWMB5
```

You've already seen these pins; they can be used as simple digital outputs with the commands `ON` and `OFF`. But these pins also have the ability to generate continuous PWM signals.

You must specify two parameters for a PWM output: frequency, and duty cycle. These are done with the commands

[1] At the present time, pin TD3 cannot be used for PWM operations. It may be used for bit I/O.

```
n  pin PWM-PERIOD
n  pin PWM-OUT
```

`PWM-PERIOD` controls the frequency of the PWM signal. (Actually, you're controlling the period, which is the reciprocal of the frequency.) This expects a value 'n' which represents ticks of a 2.5 MHz clock. You can compute the frequency of the output signal with the formula

frequency (Hz) = 2,500,000 / N

Thus a value of 2500 would give a frequency of 1 kHz. A value of 25,000 would give a frequency of 100 Hz. Alternatively, you can compute the period of the PWM signal with the formula

microseconds = N * 0.4

GOTCHA #1. For the timer output pins, `TA0` through `TD3`, you can specify a period up to 65535 decimal. This gives a frequency of about 38 Hz. But for the PWM output pins, `PWMA0` through `PWMB5`, you can only specify a period up to 32767 decimal, for a frequency of 76 Hz. This may be too fast for some PWM devices (such as RC servos). We'll see shortly how to get around this limitation.

GOTCHA #2. When you specify the period for one of the PWM output pins, you change the period for all six pins in that group (PWMA or PWMB). In other words, if you set `PWM-PERIOD` for `PWMA0`, you are *also* setting it for `PWMA1` through `PWMA5`. This is ordinarily not a problem, but you should be aware of it.

Also, you're not allowed to set the `PWM-PERIOD` to a value smaller than 256. In other words, the maximum PWM frequency is about 9766 Hz. This is to ensure that you have adequate PWM resolution when controlling the duty cycle.

`PWM-OUT` controls the duty cycle of the PWM signal, and activates the PWM output. It expects a value 'n' which is an unsigned integer in the range of 0 to 65535 (0 to FFFF hex). This corresponds to a duty cycle from 0% to 100%. So,

`0 PWMB3 PWM-OUT`	sets the duty cycle to 0% (always off),
`HEX FFFF PWMB3 PWM-OUT`	sets the duty cycle to 100% (always on),
`HEX 8000 PWMB3 PWM-OUT`	sets the duty cycle to 50% on, and
`HEX 4000 PWMB3 PWM-OUT`	sets the duty cycle to 25% on.

This is independent of the PWM frequency. A `PWM-OUT` value of `HEX 8000` will always give a 50% duty cycle, regardless of what you've specified for `PWM-PERIOD`.

You can turn off a PWM output by setting its duty cycle to zero, or by using the `OFF` command, e.g.,

```
PWMB3 OFF
```

62. Half Speed Operation

What if you need to control a bunch of RC servos with the PWMA and PWMB output pins, and they require a PWM frequency of 50 Hz? Normally, these pins can't produce anything less than 76 Hz. But there's one way to produce a slower output, and that is to *slow the entire IsoPod™ to half speed.*

The command `HALFSPEEDCPU` turns the IsoPod's master clock to half its normal 40 MHz speed. This slows *everything* in the IsoPod™ down to half speed. Instructions will run half as fast. If you specify 9600 BAUD for the serial port, you'll actually get 4800 baud. And what's most important, if you specify 100 Hz as the PWM output frequency, you'll actually get 50 Hz.

To specify 100 Hz PWM frequency, use the command

```
DECIMAL 25000 pin PWM-PERIOD
```

If you then type `HALFSPEEDCPU` you will see an output frequency of 50 Hz. (You can specify `HALFSPEEDCPU` before or after `PWM-PERIOD`, it doesn't matter. Just remember that it will also require you to change your terminal's baud rate.)

If for any reason you need to return to normal "full speed" operation, the command is `FULLSPEEDCPU`.

63. Output Polarity

For the timer output pins TA0 through TD3 *only*, you can control the polarity of the output signal.

`pin ACTIVE-HIGH`	makes the pin "active high" (the normal case). Specifying a duty cycle of 25% (hex 4000) will make the pin on for 25% of the time.
`pin ACTIVE-LOW`	makes the pin "active low." Specifying a duty cycle of 25% (hex 4000) will make the pin *off* for 25% of the time.

You can't do `ACTIVE-HIGH` or `ACTIVE-LOW` for the `PWMxx` output pins, but you don't need to. To invert the sense of the output, all you need is to do a one's complement (`INVERT`) of the value you're specifying for `PWM-OUT`. For example,

`HEX 4000 PWMB3 PWM-OUT` sets the duty cycle to 25% on, but

`HEX 4000 INVERT PWMB3 PWM-OUT` sets the duty cycle to 75% on, which is the same as setting it to 25% off.

So, you might ask, why bother? `ACTIVE-HIGH` and `ACTIVE-LOW` are really intended for PWM *input*, which will be described in the next section.

64. PWM Input

The IsoPod™ can also *measure* pulse-width-modulated (PWM) square waves on the 13 timer pins:

```
TA0 TA1 TA2 TA3
TB0 TB1 TB2 TB3
TC0 TC1
TD0 TD1 TD2
```
[2]

The commands to measure a PWM pulse width are

`pin SET-PWM-IN` to start measurement, and
`pin CHK-PWM-IN` to get the result.

`SET-PWM-IN` makes the specified timer pin an input, and puts it into the pulse-width-measurement mode. It will do nothing until it sees a rising edge on the input. Then, it will measure the time that the input is high. The falling edge after a rising edge ends the time measurement.

`CHK-PWM-IN` gets the result of the time measurement. If the rising edge has not yet been seen, or if the "high" width is still being measured (i.e., the falling edge hasn't been seen), `CHK-PWM-IN` will return a value of zero. After a complete pulse has been received (rising edge, then falling edge), the *first* use of `CHK-PWM-IN` will return the width of that pulse, which will be nonzero.

BEWARE: if you then use `CHK-PWM-IN` again, without resetting the timer, you will get an unpredictable nonzero value. Only the *first* nonzero value returned by `CHK-PWM-IN` is valid. After you receive that value, you must reset the timer with `SET-PWM-IN`.

[2] At the present time, pin TD3 cannot be used for PWM operations. It may be used for bit I/O.

The value returned by `CHK-PWM-IN` is an unsigned integer, representing ticks of a 2.5 MHz clock. This is the same timebase used for PWM output. Each tick of this clock takes 0.4 microseconds. So, the measured pulse time can be computed with the formula

microseconds = N * 0.4

If you measure a pulse input of 25000 decimal, you know that this is 10 milliseconds.

The PWM measurement has been divided into two actions ("set" and "check") to avoid the problem of Program Counter Capture. We don't want our IsoMax program to sit waiting for a pulse to be measured -- especially if the pulse never arrives! Instead we have two commands, `SET-PWM-IN` and `CHK-PWM-IN`, which are guaranteed to always execute immediately. You can test `CHK-PWM-IN` to cause a state transition when the pulse has been received.

65. Input Polarity

`SET-PWM-IN` and `CHK-PWM-IN` measure the time that the input pin is *high*. What if you need to measure the time that the input pin is *low?* This is where you need to change the polarity of the pin:

`pin ACTIVE-HIGH`	makes the pin "active high" (the normal case). The PWM-IN commands will measure the *high* duration of a pulse.
`pin ACTIVE-LOW`	makes the pin "active low." The PWM-IN commands will measure the *low* duration of a pulse.

So, actually, `SET-PWM-IN` and `CHK-PWM-IN` measure the time that the input pin is "active." `ACTIVE-HIGH` and `ACTIVE-LOW` define whether "active" is a high level or a low level.

Incidentally, note that these words aren't limited to measuring the "active" time (duty cycle) of a PWM signal. Really they measure pulse width. So they can be used to measure the width of a single pulse, too.

66. Example

Here's a simple procedural program that starts, and waits for, a PWM measurement on pin TA0:

```
: MEASURE-PWM ( -- n )
    TA0 SET-PWM-IN
    BEGIN TA0 CHK-PWM-IN ?DUP UNTIL ;
```

The key to this program is the phrase `?DUP UNTIL`. If `TA0 CHK-PWM-IN` returns a zero value, `UNTIL` will see this and continue looping. But when `TA0 CHK-PWM-IN` returns a nonzero value, `?DUP` will make an extra copy, and `UNTIL` will terminate the loop. Then the extra copy of this value is left on the stack. This way, `CHK-PWM-IN` is only called *once* with a nonzero result.

67. Analog-to-Digital Conversion

Eight pins on the IsoPod™ can be used to input analog voltages:

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

The command to read an analog value (that is, to perform an A/D conversion) is `ANALOGIN`.

`pin ANALOGIN`	Reads the given A/D input and returns its value.

`ANALOGIN` will return a result in the range 0-7FF8 hex, or 0-32760. This is actually a 12-bit A/D result that has been left-shifted 3 places, to use the full range of signed integers (0 to +32767).

A value of 32760 corresponds to an input of Vref (normally 3.3 volts). 0 corresponds to an input of 0 volts. So, the actual voltage read on the pin can be computed with the formula

$$Vin = 3.3 * N / 32760$$

68. PROCEDURAL PROGRAMMING

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

The IsoMax procedural language is very similar to the programming language Forth. There are some significant changes because of the Finite State Machine functions, the Object-Oriented functions, and the architecture of the IsoPod™ processor (which has separate data and program memories, and can't address memory as bytes). But if you are familiar with Forth, and particularly Max-Forth, you can skip most of this section.

69. The Dictionary

All of the commands and operations known to IsoMax are kept in a "dictionary." You will frequently hear these commands and operations called "words," because they are the words in the IsoMax language. In fact, you can print out a list of all the words known to IsoMax with the command

```
WORDS
```

This just prints a list of the words. Their definitions are much longer! You can find short definitions for all these words in section 18 of this manual (IsoMax Glossary).

Possible point of confusion: Don't confuse a "word" in the language, with a "word" of memory. A memory word (on the IsoPod™) is 16 bits of storage. A word of the language can be any symbol made of non-blank characters. When there is risk of confusion, we will generally refer to 16 bits of storage as a "cell" of memory.

70. The Stacks

Numbers, addresses, and data which is being operated upon are normally held on *stacks.* Like the name implies, when you put something on a stack, it becomes the topmost item on the stack, and everything that was already on the stack is effectively "pushed down" one deeper. Likewise, when you take something off the top of the stack, the stuff underneath it "pops up" one position, and what was the second item on the stack becomes the new top item on the stack.

For the most part, the operation of stacks is invisible and automatic. One visible effect (which will be familiar to owners of Hewlett-Packard calculators) is that arithmetic operations require you to place the two operands on the stack first, and then specify the operation to be performed. In other words, instead of saying

```
2 + 3
```

in IsoMax you would say

```
        2 3 +
```

We'll see more examples of this shortly.

There is one stack for fixed-point data (including integers, characters, and addresses), and a second stack for floating-point numbers. You don't need to specify this – each operation automatically uses the correct stack. But you might need to be aware of the two different stacks, if you're moving values to and from them.

71. Stack Notation

Whenever you type a number, IsoMax puts it on the stack. (Integers on the integer stack, floating-point numbers on the floating-point stack.) If you type a second number, it gets put on the stack and the previous number gets pushed down. So, for instance, if you type

```
        1 2 3
```

the "3" will be the topmost item on the stack, "2" will be under it, and "1" will be on the bottom. You can see this with the command .S ("print stack"). Try typing the command

```
        1 2 3 .S
```

and you will see how the numbers are "stacked" (with the "3" on the top).

When we are describing parameters to be put on the stack, or values which are placed on the stack, we will use this left-to-right notation. The rightmost item in the description corresponds to the topmost item on the stack. So when you see parameters

```
        a b c
```

you know it really means

```
        c
        b
        a
```

with "c" on the top of the stack.

72. Arithmetic Operations

Most of the time you'll be using 16-bit integers. These can be treated as unsigned numbers, in the range 0..65535, or as signed numbers, in the range –32768..+32767. The four basic arithmetic operators are add, subtract, multiply, and divide:

```
        10 2 +      adds 10 and 2, giving 12
```

```
10 2 -      subtracts 10-2, giving 8
10 2 *      multiplies 10 by 2, giving 20  (signed numbers)
10 2 /      divides 10 by 2, giving 5  (signed numbers)
```

Notice the order of the operands for subtract and divide. This is easy to remember, because it's the same left-to-right order you would use if you were writing these as algebraic equations. That is,

```
10 2 -      performs the computation    10 – 2
10 2 /      performs the computation    10 / 2
```

If you want to see the result of these computations, you can use the `.S` command. Or, you can use the `.` command (just a period character), which prints the topmost stack item and removes it from the stack. For example, try

```
10 3 / .
```

and you should see the result 3. Why 3 and not 3.33333? We're using integer math, so we get "3 with a remainder of 1," not 3.33333. To see the remainder. use the MOD command:

```
10 3 MOD .
```

When the IsoPod™ starts running, it expects decimal (base 10) numbers. But you can change this at any time. If you type the command `HEX`, all numbers from that point on will be entered and printed in hexadecimal (base 16). To change back to base 10, type the command `DECIMAL`. You can use this with the . (print) command to perform simple base conversions. For example:

```
HEX A0 DECIMAL .
```

will print 160, the decimal equivalent of A0 hex.

You can type negative numbers, like –12 or even –FFF (in hexadecimal base). If you want to negate the result of a computation, you can use the command `NEGATE`. For example,

```
-12 3 * NEGATE .
```

will print 36, because –12 times 3 is –36, and the negative of –36 is 36.

73. *Floating-Point Operations*

To type a floating-point number into IsoMax, you *must* include an exponent, in the form Enn, as the suffix of the number. For example, all of these represent a floating-point value of 2.0:

```
2.0E0
2.000E0
2E0
0.2E1
.2E01
20.E-1
```

The "E" followed by a (positive or negative) number is required. The following will *not* work:

```
2.0
2.
002
2.000
```

Also, you *must* be in DECIMAL base to type a floating point number. This is because "E" is a valid hexadecimal digit. So,

`DECIMAL 2E0`	gives the floating-point value 2.0, but
`HEX 2E0`	gives the integer 2E0 hex (736 decimal)

You can use the command `F.S` to display the contents of the floating-point stack. The command `F.` will print (and remove) the topmost item on the floating-point stack in a "fixed point" notation, and the command `E.` will print (and remove) the topmost item in an "exponential" notation. To see this, try

```
DECIMAL 1.E1 2.E1 3.E1
F.S
F.
E.
E.
```

Floating-point arithmetic operations are similar to the integer operations, but have the prefix "F". Here are add, subtract, multiply, and divide:

`10.E0 3.E0 F+`	adds 10.0 and 3.0, giving 13.0
`10.E0 3.E0 F-`	subtracts 10.0-3.0, giving 7.0
`10.E0 3.E0 F*`	multiplies 10.0 by 3.0, giving 30.0
`10.E0 3.E0 F/`	divides 10.0 by 3.0, giving 3.3333

Other floating-point operations include trigonometric and transcendental functions. The complete list can be found in the glossary, Section 18.4.

74. Variables

Because IsoMax carries out computations on its stacks, you very rarely need to use "variables" such as X or Y or VELOCITY or VOLTAGE. But sometimes you do need to store a value between computations. So, IsoMax allows you to have named variables.

You must define a variable before you use it. This is done with the command `VARIABLE` (integer) or `FVARIABLE` (floating-point):

Integer	**Floating-point**
`VARIABLE name`	`FVARIABLE name`

In either case, "name" is a name you choose for the variable. This can be any combination of up to 31 non-blank characters. Even special characters and punctuation may be freely used. For example, the following are all valid variable names:

```
X  Y1  Velocity  $PROFIT  $  4TH_SPEED  %#@!
```

Names can begin with numbers, and can be entirely non-alphabetic characters. Two restrictions, though. First, don't use a name that's already in use by IsoMax (as you can see with `WORDS`) . This will cause confusion. IsoMax will allow it, but will warn you by telling you that your name is "not unique."

Second, don't use a name that's all numbers. IsoMax will allow that, and *won't* warn you, and then when you type that number, you'll get the variable instead of the number. As you can imagine, this will lead to no end of confusion. Be sure that all your names have one non-numeric character (and remember that A through F are digits in hex)!

You can use upper or lower case in your names, but remember that IsoMax is case-sensitive. `VELOCITY`, `VELocity`, and `Velocity` are all different names.

When you have defined a variable, you can store a value into that variable with the ! or F! commands. You can fetch the stored value with the @ and F@ commands.

Integer	**Floating-point**	
`name !`	`name F!`	stores a value in variable "name"
`name @`	`name F@`	fetches a value from variable "name"

This is not like other languages, which let you use just the name of a variable in place of a number in an equation. To get the value of a variable, you *must* use `@` (for integer variables) or F@ (for floating-point variables). So, if you want to multiply (floating-point) Principal by Interest to compute a payment, you'd have to type

```
Principal F@  Interest F@  F*  F.
```

Of course, before you could do this you would have had to define the variables

```
FVARIABLE Principal
FVARIABLE Interest
```

and you would have had to store some values into these variables, e.g.,

```
10000.E0 Principal F!
0.05E0 Interest F!
```

75. Accessing Memory and I/O

When you define a VARIABLE, what you're really doing is reserving a memory location and giving that memory location a name. The operators @ and ! fetch from a memory location, and store to a memory location, respectively.

You can use @ and ! with *any* memory locations, not just variables. Suppose that you know a value is stored at memory address $6A2. You can get that value with

```
HEX 6A2 @
```

Suppose you want to store a value of $1234 into that location. You can use

```
HEX 1234 6A2 !
```

When might this be useful? Most of the time, you'll want to use named VARIABLEs, because a variable will always be placed in an unused part of memory. If you try to choose memory addresses yourself, you might choose an address that's being used by IsoMax for something else.

But there is one situation when you might want to read or write a known memory location. The input and output of the IsoPod's DSP56F805 CPU is *memory-mapped.* This means that, instead of accessing the peripherals with IN and OUT instructions, you use normal memory fetch and store instructions. So, @ and ! give you access to the complete I/O capability of the IsoPod processor!

For example, the Port A Data Register is located at address $0FB1. The Port A Data Direction Register (DDR) is located at address $0FB2, and the Peripheral Enable Register (PER) is at $0FB3. Writing zero to both the DDR and PER will make Port A an input port, and you can then read the Data Register to read the eight input pins.

```
HEX
     0 0FB2 !
0 0FB3 !
0FB1 @
```

This is not for the inexperienced user. There are *no* restrictions on `@` and `!`, so it's quite possible for you to lock up the IsoPod completely by writing the wrong value to the wrong location. You should refer to Motorola's *DSP56F801/803/805/807 16-Bit Digital Signal Processor User's Manual* for a complete description of the on-chip I/O of the DSP56F805 processor, and its memory addresses.

Besides, isn't this easier?

```
PORTA GETBYTE
```

ServoPod owners note: The ServoPod uses the DSP56F807 processor, which has *different* I/O addresses from the IsoPod's DSP56F805.

76. Logical Operations

IsoMax also lets you perform logical operations on 16-bit values. The four basic arithmetic operators are AND, OR, XOR, and INVERT:

`HEX 3A 0F AND`	bitwise logical AND, giving 0A
`HEX 3A 0F OR`	bitwise logical OR, giving 3F
`HEX 3A 0F XOR`	bitwise exclusive OR, giving 35
`HEX 3A INVERT`	bitwise inversion, giving FFC5

Notice that `INVERT` takes only one parameter. Also, all of the logical operations act on 16 bit values. If you try to print the result with the . operator, you may be surprised:

```
HEX 3A INVERT .      prints -3B
```

This is because FFC5 is a negative number in two's complement notation, and `.` prints signed numbers. To print unsigned numbers, use the `U.` ("unsigned print") operator:

```
HEX 3A INVERT U.     prints FFC5
```

The logical operations are is especially useful when you're working with I/O, when you need to act on specific bits. For example, suppose you need to read the low 4 bits of port A as a hex number from 0 to F. You could read the four bits individually, and write some code to merge them into a 4-bit value. But it's much easier to say

```
PORTA GETBYTE HEX 0F AND
```

which reads all 8 bits of the port, and then "masks off" the unwanted bits.

What if you need the *high* 4 bits of the port? It's only an 8-bit port, so the "mask" should be F0 instead of 0F. You then need to shift the bits "down" four places:

```
    PORTA GETBYTE HEX F0 AND  2/ 2/ 2/ 2/
```

The `2/` operator ("two-divide") gives a one-bit right shift. For integers, this is equivalent to dividing by two, hence the name. Applying it four times gives a total of four shifts to the right.

The corresponding left-shift operator is `2*` ("two-times").

77. Adding New Definitions

Much of the time you will be using the IsoMax operations interactively -- as we have seen above -- or in the `CONDITION` or `CAUSES` phrases of an IsoMax state machine. What you have learned so far is sufficient for these uses, although you'll probably want to look at the IsoMax Glossary in Section 18 to see the full range of operations which are available to you.

But there may come a time when you want to create a procedural subroutine. This might be because

- There's a complex function you perform frequently, and you're tired of typing it all the time, or
- You want to write a computer program (in the traditional sense) and commit it to the IsoPod's memory.

In either case, you do this by adding a new word to the IsoMax dictionary. This new word will contain your complex function or your application program.

Just like with an English dictionary, you add a new word by first giving the name of the new word, and then defining that word *using only words which are already known.* IsoMax marks the start and end of a new definition with `:` and `;` as follows:

```
    : name-of-new-word    ...definition...  ;
```

The spaces after `:` and before `;` are required. The name of the new word can be any combination of up to 31 non-blank characters, just like `VARIABLE` names.

Let's go back to our recent example, and assume that a 4-bit DIP switch is connected to the high 4 bits of Port A. We know how to read this port, mask the bits, and shift them to the low 4 bits. But we're going to be doing this a lot, and we don't want to type that long phrase every time. Also, to make the code more readable (and more maintainable), we'd like to call it something meaningful like `GET-DIP-SWITCH`. Here's how you can do it:

```
HEX
: GET-DIP-SWITCH  PORTA GETBYTE F0 AND
  2/ 2/ 2/ 2/ ;
```

Here we are telling IsoMax to add a new word, `GET-DIP-SWITCH`, to the dictionary. The "definition" of this new word is `PORTA GETBYTE HEX F0 AND 2/ 2/ 2/ 2/`. What this means in practice is that, whenever IsoMax sees `GET-DIP-SWITCH`, it will perform the action `PORTA GETBYTE HEX F0 AND 2/ 2/ 2/ 2/`. (Strictly speaking, we've created a *subroutine* containing those IsoMax instructions.)

Remember that IsoMax is *free-format* so you can split the definition across multiple lines, and use spaces to indent. **TAKE NOTE:** if you're going to use numbers inside the definition, you must specify the number base *outside* the definition. In this example, we put `HEX` before we started the definition.[3]

`GET-DIP-SWITCH` will have exactly the same stack effect as its definition. Since `PORTA GETBYTE HEX F0 AND 2/ 2/ 2/ 2/`. leaves a single value on the stack, `GET-DIP-SWITCH` will leave a single value on the stack.

Of course, we could also have written something which takes values from the stack. Maybe we need a four-bit right shift frequently:

```
: RIGHT-SHIFT-4   2/ 2/ 2/ 2/ ;
```

This will take a value on the stack, shift it right four times, and then leave the result on the stack. So you see, the stack is how we pass values to a function, and how we get results from a function. These are the *input parameters* and *output parameters* of the function.

Well, once we've told IsoMax what RIGHT-SHIFT-4 means, why can't we use that to define GET-DIP-SWITCH? We can:

```
: RIGHT-SHIFT-4   2/ 2/ 2/ 2/ ;
HEX
: GET-DIP-SWITCH  PORTA GETBYTE F0 AND RIGHT-SHIFT-4 ;
```

78. Removing definitions

If you've been typing this example in, you've probably seen the warning `GET-DIP-SWITCH NOT UNIQUE`. This is IsoMax telling you that you've defined a word twice in its dictionary. This won't break IsoMax, but it will cause you some confusion, since you won't necessarily know what definition is being used at any given time.

[3] If you put `HEX` inside the definition, that number base won't take effect until later, when you execute `GET-DIP-SWITCH`. This is sometimes useful, but usually is not what you want.

It's better for all concerned if you tell IsoMax to forget your previous definition of the word. You do this with the command

```
FORGET GET-DIP-SWITCH
```

This gets rid of the old definition, and leaves you free to start a new one. (Strictly speaking, `FORGET` gets rid of the *most recent* definition of the word. If you've defined the word twice, you'll need to use `FORGET` twice to get rid of both definitions.)

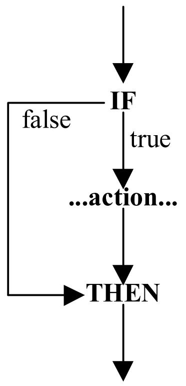
TAKE NOTE: `FORGET` will not just forget the word you specify, it will forget *all words you have defined since that word*. In the last example, if you had typed `FORGET RIGHT-SHIFT-4`, you would *also* lose the definition of `GET-DIP-SWITCH`. This can be useful: if you've written a few dozen words, and you want to forget them all (so you can start over), you don't need to type a few dozen `FORGET` commands. Just forget the first word, and all the following words will go too.

79. Program Control

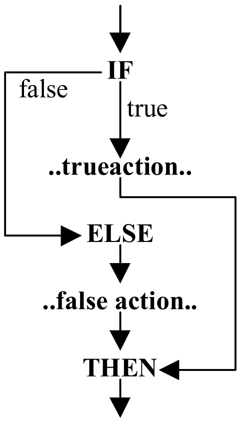
We've seen how to write subroutines (as new word definitions), how to do arithmetic and logical functions, how to store data in memory variables, and how to do I/O. There's still one thing missing before we can write any computer program: how do we perform actions *conditionally*? That is, how do we change the *flow of control* of the program, based upon an input or the result of a calculation?

IsoMax offers six different constructs for program control. These correspond to the basic *control structures* from the discipline of structured programming.

`IF ...some action... THEN`



This performs an action *if* some condition is true. The condition is given by a value on the stack when `IF` is encountered. A zero value is "false", and any nonzero value will be considered "true." A true value causes the code between `IF` and `THEN` to be performed. A false value causes that code to be skipped.

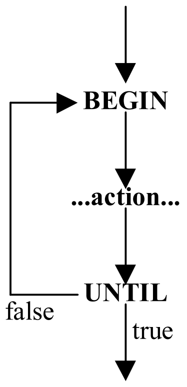


`IF ..true action.. ELSE ..false action.. THEN`

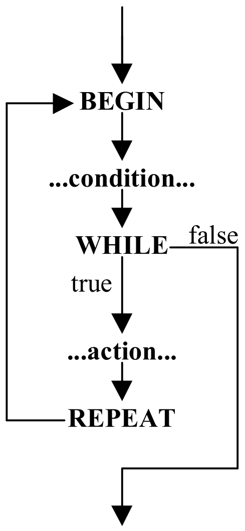
This is similar to `IF..THEN`, except that it performs one action if the condition is true (nonzero), and a different action if the condition is false (zero). **Remember** that, unlike some other languages, `THEN` terminates the control structure. The code following then is always executed.

`BEGIN ...some action... UNTIL`

This performs an action repeatedly *until* some condition is true. The condition is given by a value on the stack when `UNTIL` is encountered. A zero (false) value means "do the action again," and will cause a loop from the `UNTIL` back to the `BEGIN`. A nonzero (true) value means "terminate the loop," and will cause execution to continue on to the code after the `UNTIL`. Note that it is the action inside the loop that produces the true/false value for `UNTIL`! Whatever else is done, this action *must* include code which leaves this "exit/loop" value on the stack. Note also that the action inside the loop will always be performed at least once!



`BEGIN ...condition... WHILE ...some action... REPEAT`

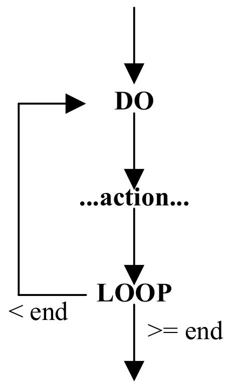


This performs an action repeatedly *while* some condition is true. This is similar to `BEGIN..UNTIL` with the following differences:

- The code that produces the "exit/loop" value is placed *before* the `WHILE`, and the action to be taken is *after* the `WHILE`.
- A zero (false) value means "terminate the loop"; a nonzero (true) value means "do the action and keep looping."
- It's possible for the action to be performed *zero* times.

Another way to look at this: a false value at the `WHILE` will cause the program to immediately jump to the code after the `REPEAT` (thus exiting the loop). A true value at the `WHILE` will cause the code immediately following the `WHILE` to be executed, and then `REPEAT` will loop back to `BEGIN`.

`end start DO  ...some action... LOOP`



This performs an action repeatedly for a given number of times. This loops over values from "start" to "end-1". The "end" and "start" values are given on the stack when `DO` is encountered, with the "start" value on the top of stack, and the "end" value second on the stack. These values can be determined from a computation, but often will just be numeric constants. For example

```
10 0 DO  I .  LOOP
```

will perform the action `I .` ten times, with the *loop index* going from 0 to 9 (inclusive). The operator `I` will always return the value of the current loop index. (Unlike other languages, you don't need to use a variable for this.) So in this example, the action `I .` will print the loop index on each pass through the loop.

`end start DO  ...some action... n +LOOP`

This is similar to `DO..LOOP` except that the loop index is incremented by "n" instead of 1. "n" is the value on top of the stack when `+LOOP` is encountered; it is usually a constant, but could be the result of a computation. It may be positive or negative. If the increment is negative, "end" must be less than "start," and the loop will proceed all the way to the end value (not end-1).

IMPORTANT LIMITATION: All of these control structures can be used inside a word definition, and inside a `CONDITION` or `CAUSES` phrase in an IsoMax state machine. But they *can not* be used interactively from the command line.

80. DO Loop Example

It's instructive to write some simple definitions which show how `DO` loops work:

```
DECIMAL
: TEST#1   10 0 DO I . LOOP ;
TEST#1
```

This will just print the value of the loop index as it goes from 0 to 9.

```
: TEST#2   DO I . LOOP ;
10 0 TEST#2
30 20 TEST#2
```

This has the same action as `TEST#1`, but instead of "hard coding" the loop limits inside the definition, we are passing them as parameters on the stack. So we can try the loop with a number of different start and end parameters.

```
: TEST#3  DO I . 3 +LOOP ;
10 0 TEST#3
```

This illustrates an increment greater than 1. You can try different end values -- say, 11, 12, and 13 -- and you'll see that the loop always stops short of the end value.

```
: TEST#4   DO I . -2 +LOOP ;
0 10 TEST#4
```

This illustrates a negative increment. You'll see that the loop *will include* the end value if it can, but it will not go past it. (Try end values of 1 and -1 instead of 0.)

Remember: The most common mistake made with `DO` loops is to get the order of the start and end values backwards. The "start" value is the last thing put on the stack.

81. Comparisons

Now that you have the ability to change the flow of your program based on a condition, you need some operators to create those true or false flags. IsoMax has four operators which will let you *compare* two numbers:

`a b =`	returns true if a=b
`a b >`	returns true if a>b (signed numbers)
`a b <`	returns true if a<b (signed numbers)
`a b U<`	returns true if a<b (*unsigned* numbers)

Here "a" and "b" refer to any two numbers on the stack. "b" is on top of the stack, exactly as though you had typed the numbers "a b" at the keyboard.

You'll note that there isn't a `U>` operator. We'll see in the next section how to construct one. There also isn't a `U=` operator, since = works for both signed and unsigned numbers.

IsoMax also has three operators which let you examine a single number:

`a 0=`	returns true if a is zero
`a 0>`	returns true if a is greater than zero
`a 0<`	returns true if a is less than zero (negative)

Finally, there is an operator which will turn true to false, and false to true:

```
a NOT
```
logical inverse of a

Do not confuse `NOT` with `INVERT`. `INVERT` is a bitwise operator, which individually inverts all 16 bits of a value on the stack. `NOT` is a logical operator, which returns true (a nonzero value, actually $FFFF) if the value on the stack is false (zero), and returns false (zero) if the value on the stack is true (any nonzero value). `INVERT` is for bits. `NOT` is for true/false values.

Of course, these comparisons are also useful in IsoMax state machines. The phrase between `CONDITION` and `CAUSES` must leave a true/false value on the stack. How this value is produced is up to you; it might come from testing an input bit, or it might come from comparing two numeric values.

82. *Stack Operations*

For most short pieces of code, like IsoMax `CONDITION` and `CAUSES` phrases, you'll only have one or two things on the stack, and they'll be in the right place at the right time. But sometimes values get put on the stack in the wrong order, or you have an extra value that you don't need, or maybe you need a value twice. To handle these little details, IsoMax provides an assortment of stack operators.

Another word about stack notation: recall that we use the notation

```
a b c
```

to signify that there are three values on the stack, with "a" on the bottom, "b" in the middle, and "c" on the top. (This is how they'd be on the stack if you typed three numbers on the command line, in the same left to right order.) Since the stack operators rearrange the values on the stack, we need "before" and "after" pictures to illustrate their operation. The common notation for this is

```
a b c --- x y z
```

where "a b c" is the stack data *before* the operation, and "x y z" is the stack data *after* the operation. We'll see how this works in a moment.

First, let's look at words that get rid of items on the stack:

Word	**Stack effect (before --- after)**
`DROP`	`a ---`
`2DROP`	`a b ---`

`DROP` simply takes whatever is on top of the stack, and gets rid of it. `2DROP` gets rid of the top two items on the stack. Anything else that may be deeper on the stack is unaffected, so it's not shown in the stack notation. So, if you have 1 2 3 on the stack and you do a `DROP`, you'll wind up with 1 2 on the stack...only the 3 is `DROP`ped.

There are also words to duplicate items on the stack:

```
DUP   a --- a a
OVER a b --- a b a
```

`DUP` just takes whatever is on top of the stack, and makes a second copy of it on the stack (so then you have two of them). You might want to do this if you need to test if a value is zero or nonzero, and then if it's nonzero, perform some computation with it.

`OVER` is trickier: it makes a copy of the *second* item on the stack, and pushes that copy onto the top of the stack (pushing everything else down). If that sounds confusing, just remember that `OVER` takes "a b" and gives you "a b a".

To rearrange the values on the stack, you can use:

```
SWAP a b --- b a
ROT   a b c --- b c a
```

`SWAP` just "swaps" the top two stack items (the second becomes first, and the first becomes second). `ROT` is short for "rotate"; it rotates the top *three* items on the stack, such that the deepest item becomes the topmost item. If you do three rotates -- that is, `ROT ROT ROT` -- you'll get right back to where you started.

IsoMax has many more stack operators, and you're encouraged to look at the IsoMax Glossary in Section 18 to learn about more of them. But these six will handle the most of the manipulations you'll need to perform.

83. Example

Remember that we don't have a `U>` operator? It ought to look like this:

`a b U>`   should return true if a>b (unsigned numbers)

Well, if a>b, then it follows that b<a. And we *do* have an operator for unsigned-less-than (`U<`). So all we need to do is reverse the order of a and b, and then use `U<`:

`a b SWAP U<`   returns true if b<a, and thus if a>b

We can take this one step further, and use our ability to make definitions to add this as a new part of the IsoMax language!

```
: U>  SWAP U< ;
```

This defines a new word named `U>` which does the same thing as `SWAP U<`. Presto! What's nice is that this new word becomes a part of the language, just like all the words originally "known" to IsoMax. Any place you could use the built-in word `U<`, you can use your new word `U>`. There is no distinction between "built-in operators" and "user functions"; you can add new comparison, arithmetic, and logical operators as you please. For this reason, IsoMax is called an *extensible* language -- you can add new language elements at any time.

84. Word list

The complete word list is found in the IsoMax Glossary (Section 18) at the end of this manual.

85. ADVANCED PROGRAMMING

86. IsoMax v0.3 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.

87. IsoMax v0.6 Memory Map

DATA MEMORY

0000 0245	Data RAM (Kernel)
0246 07FF	Data RAM (User)
0800 0BFF	reserved
0C00 0FFF	peripherals
1000 17FF	Data Flash (SAVE-RAM)
1800 1FFF	Data Flash (User)

PROGRAM MEMORY

0000 13FF	Program Flash (Core)
1400 1FFF	Program Flash (User)
2000 3FFF	Program Flash (User) *'803 and* *'805 only*
4000 7DFF	Program Flash (Kernel)
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.

88. IsoMax v0.6 Memory Map – DSP56807

DATA MEMORY

0000 0245	Data RAM (Kernel)
0246 0FFF	Data RAM (User)
1000 17FF	peripherals
1800 1FFF	reserved
2000 2FFF	Data Flash (SAVE- RAM)
3000 3FFF	Data Flash (User)

PROGRAM MEMORY

0000 13FF	Program Flash (Core)
1400 3FFF	Program Flash (User)
4000 7DFF	Program Flash (Kernel)
8000 EFFF	Program Flash (User)
F000 F7DF	Program RAM (User)
F7E0 F7FF	Program RAM (Kernel*)

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

89. Starting IsoMax State Machines

When the IsoPod is reset, it disables all running state machines. You must explicitly start your state machines as part of your application -- usually, in your autostart code. There are two ways to do this: with `INSTALL`, or with `SCHEDULE-RUNS`.

90. Using INSTALL to start a State Machine

From IsoMax version 0.36 onward, the preferred method of starting state machines is with `INSTALL`. After you have defined a state machine, you can start it by typing

```
state-name SET-STATE
INSTALL machine-name
```

Note that you must use `SET-STATE` to specify the starting state of the machine **first.** This is because `INSTALL` will start the machine immediately. To start more machines, simply `INSTALL` them one at a time:

```
state-name-2 SET-STATE
INSTALL machine-name-2
state-name-3 SET-STATE
INSTALL machine-name-3
etc.
```

Normally,[4] the state machine will start running immediately at the default rate of 100 Hz. `SET-STATE` and `INSTALL` can be used even while other state machines are running, that is, `INSTALL` will *add* a state machine to an already-running list of state machines.

At present, up to 16 state machines can be `INSTALL`ed. Attempting to `INSTALL` more than 16 machines will result in the message "Too many machines." To install more machines, you can use `UNINSTALL` or define a `MACHINE-CHAIN` (both described below).

`SET-STATE` and `INSTALL` can be used interactively from the command interpreter, or as part of a word definition.

91. Removing a State Machine

`INSTALL` builds a list of state machines which are run by IsoMax. `UNINSTALL` will remove the last-added machine from this list. You can use `UNINSTALL` repeatedly to remove more machines from the list, in a last-in first-out order. For example:

[4] The commands `COLD`, `SCRUB`, and `STOP-TIMER` will halt IsoMax. The command `SCHEDULE-RUNS` will override the `INSTALL`ed state machines and dedicate IsoMax to running a particular machine chain.

```
INSTALL machine-name-1     ( SET-STATE commands have been omitted
for clarity)
INSTALL machine-name-2
INSTALL machine-name-3
     . . .
UNINSTALL              ...removes machine-name-3
UNINSTALL              ...removes machine-name-2
UNINSTALL              ...removes machine-name-1
UNINSTALL              ...removes nothing
```

If there are no state machines running, `UNINSTALL` will simply print the message "No machines."

To remove *all* the `INSTALL`ed state machines with a single command, use `NO-MACHINES`.

92. Changing the IsoMax Speed

When the IsoPod is reset, IsoMax returns to its default rate of 100 Hz -- that is, all the state machines are performed once every 10 milliseconds. You can change this rate with `PERIOD`. The command

```
n PERIOD
```

will set the IsoMax period to "`n`" cycles of a 5 MHz clock. Thus,

```
DECIMAL 5000 PERIOD ...will execute state machines once per millisecond

DECIMAL 1000 PERIOD ...will execute state machines every 200
microseconds
```

...and so on. You can specify a period from 10 to 65535.[5] (Be sure to specify the `DECIMAL` base when entering large numbers, or you may get the wrong value.) The default period is 50000.

93. Stopping and Restarting IsoMax

Certain commands will halt IsoMax processing:

the `COLD` command
the `SCRUB` command

This is necessary because either `COLD` or `SCRUB` can remove state machines from the IsoPod memory.[6] You can also halt IsoMax manually with the command `STOP-TIMER`.

[5] Note, however, that very few state machines will be able to run in 2 microseconds (corresponding to `10 PERIOD`). If you specify too small a `PERIOD`, no harm will be done, but IsoMax will "skip" periods as needed to process the state machines.

In all these cases, the timer that runs IsoMax is halted. So, even if you `INSTALL` new state machines, they won't run. To restart IsoMax you should use the command `ISOMAX-START`. This command will

a) Remove all installed state machines, and
b) Start IsoMax at the default rate of 100 Hz.

Since `ISOMAX-START` removes all installed state machines, you must use it *before* you use `INSTALL`. For example:

```
STOP-TIMER
     . . .
ISOMAX-START
state-name-1 SET-STATE
INSTALL machine-name-1
state-name-2 SET-STATE
INSTALL machine-name-2
state-name-3 SET-STATE
INSTALL machine-name-3
```

Resetting the IsoPod does the same as `ISOMAX-START`: it will remove all installed state machines, and reset the timer to the default rate of 100 Hz.

94. Running More Than 16 Machines

`INSTALL` can install both state machines and *machine chains*. A "machine chain" is a group of state machines that is executed together. Machine chains, like state machines, are compiled as part of the program:

```
MACHINE-CHAIN chain-name
     machine-name-1
     machine-name-2
     machine-name-3
END-MACHINE-CHAIN
```

This example defines a chain with the given name, and includes the three specified state machines (which must already have been defined). A machine chain can include any number of state machines.

You must still set the starting state for each of the state machines in a machine chain, before you install the chain. So, you could start this example chain with:

```
state-name-1 SET-STATE        ...a state in machine-name-1
state-name-2 SET-STATE        ...a state in machine-name-2
state-name-3 SET-STATE        ...a state in machine-name-3
```

[6] The command `FORGET` can also remove state machines from memory. Be very careful when using `FORGET` that you don't remove an active state machine; or use `STOP-TIMER` to halt IsoMax first.

```
    INSTALL chain-name
```

You can of course `UNINSTALL` a machine chain, which will stop all of its state machines.

95. Using SCHEDULE-RUNS

Prior to IsoMax version 0.36, the preferred method of starting state machines was with `SCHEDULE-RUNS`.[7] `SCHEDULE-RUNS` worked only with machine chains, and required you to specify the IsoMax period when you started the machines:

```
    EVERY n CYCLES SCHEDULE-RUNS chain-name
```

`SCHEDULE-RUNS` is still available in IsoMax, to allow older IsoMax programs to be compiled. **However,** you should be aware that using `SCHEDULE-RUNS` will *disable* any machines started with `INSTALL`. `SCHEDULE-RUNS` *replaces* any previously running state machines -- including any previous use of `SCHEDULE-RUNS` -- and there is no "uninstall" function for it. After using `SCHEDULE-RUNS`, the only ways to "reactivate" the `INSTALL` function are

a) use the `ISOMAX-START` command, or
b) reset the IsoPod

`ISOMAX-START` will disable any machine chain started by `SCHEDULE-RUNS`, and will re-initialize IsoMax. You can then `INSTALL` state machines as described above.

You can use the `PERIOD` command to change the speed of a machine chain started with `SCHEDULE-RUNS`.

96. Autostarting State Machines

When the IsoPod is reset, all state machines are halted. (Strictly speaking, the IsoMax timer is running, but the list of installed state machines is empty.) To automatically start your state machines after a reset, you must write an autostart routine, which uses `SET-STATE` and `INSTALL` to start your machines. For example:

```
    : MAIN
         state-name-1 SET-STATE
         INSTALL machine-name-1
         state-name-2 SET-STATE
         INSTALL machine-name-2
         state-name-3 SET-STATE
         INSTALL machine-name-3
```

[7] Some versions of IsoMax prior to version 0.36 have a different implementation of `INSTALL`. That implementation does not work as described here, so for those versions of IsoMax we recommend you use `SCHEDULE-RUNS`.

```
          . . . more startup code . . .
          . . . application code . . .

     ; EEWORD

     SAVE-RAM
     HEX 7C00 AUTOSTART MAIN
```

In this example, the word `MAIN` is executed when the IsoPod is reset. The first thing it does is to install three state machines. Note that these machines will begin running immediately. If you need to do some initialization before starting these machines, that code should appear before the first `INSTALL` command.

Refer to "Autostarting an IsoMax Application" for details about using `SAVE-RAM` and `AUTOSTART`.

97. IsoMax State Machine Language Reference

This illustrates the different options for defining state machines, states, and state transitions.

98. Defining State Machines

A state machine is defined by name:

```
MACHINE <name-of-machine>
```

If the machine will be moved to Flash ROM, the MACHINE declaration must be immediately followed by EEWORD:

```
MACHINE <name-of-machine> EEWORD
```

99. Defining States

Once a state machine has been defined, all of the states which will be part of that machine must be named:

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

The last example above illustrates a debugging option which is available for states. If WITH-VALUE ... AT-ADDRESS are specified, the value 'n' will be stored at address 'a' when a transition is made *to* this state.[8]

If the state machine will be moved to Flash ROM, *each* state declaration must be immediately followed by EEWORD, thus:

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

100. Defining States

After the states have been named, the transitions between the states can be defined:

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

[8] This value is actually stored by either `TO-HAPPEN`, `THIS-TIME`, or `NEXT-TIME`, when they are used by another state to select this as the new state. The tag value is not stored when `SET-STATE` is used.

```
IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> THIS-TIME

IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> NEXT-TIME
```

`<boolean computation>` must be a fragment of procedural (Forth) code which leaves a true/false (nonzero/zero) condition on the stack. If the result of this computation is true, the actions following CAUSES (a compound action and a new state) will be performed.

`<compound action>` is an optional fragment of procedural (Forth) code which is performed when the transition condition is satisfied, and before the state transition actually takes place. This must be stack-neutral (the completed action must take nothing from, and leave nothing on, the stack).

Usually when a transition is made to a new state, that state will be evaluated -- that is, all of its `CONDITION` clauses will be examined -- on the *next* IsoMax cycle. This is the safest approach, and ensures that all state machines receive adequate service. This is what happens when you specify the next state `TO-HAPPEN` or `NEXT-TIME`. (`TO-HAPPEN` is a synonym for `NEXT-TIME`).

There may be very special cases when it is important to evaluate the new state *immediately* upon a transition to that state. To achieve this you specify the next-state-name `THIS-TIME`. This is a hazardous practice, however, since it's very easy to construct a loop of states that will never terminate. `THIS-TIME` is strongly discouraged, and should only be used when absolutely necessary, and with great care.

If the state machine will be moved to Flash ROM, *each* transition definition must be immediately followed by **IN-EE** (*not* EEWORD), thus:

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

IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> THIS-TIME IN-EE

IN-STATE <parent-state-name>
     CONDITION <boolean computation>
     CAUSES <compound action> THEN-STATE <next-state-name> NEXT-TIME IN-EE
```

101. Defining Input Conditions

Often the boolean condition in a state transition will simply involve testing an input pin, an I/O register, or a memory location for a bit to be set or cleared. To make this programming easier, you can define an *input trinary*:

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

The trinary must be given a name. This name acts like a subroutine: when it is used, the trinary tests the value at the specified address, and returns a true/false result. To be precise: the value at address "a" is fetched, and logically ANDed with the `TEST-MASK`. This result is logically XORed with the `DATA-MASK`. If the result is nonzero, a true flag is left on the stack; if the result is zero, a false flag is left.

You should think of this as follows: the `TEST-MASK` specifies which bit is of interest. `DATA-MASK` specifies an optional inversion. Although these will usually act on a single bit, you can certainly have masks with multiple bits. Just remember that if *any* bit in the AND/XOR result is nonzero, the result will be logically "true."

`TEST-MASK`, `DATA-MASK`, and `AT-ADDRESS` can be specified in any order. So, the following are equivalent:

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

Input trinaries can be used in state transitions and in procedural (Forth) code. **You are not required to use trinaries for the boolean computation in a state transition.** They are merely provided as a convenience.

An input trinary can be moved to Flash ROM with EEWORD:

```
DEFINE <name>  TEST-MASK <n>  DATA-MASK <m>  AT-ADDRESS <a>  FOR-INPUT
EEWORD
```

102. Defining Output Actions

Many actions involve setting or clearing a bit in an I/O register or a memory location. To make this programming easier, you can define an *output trinary* in one of two forms:

```
DEFINE <name>  SET-MASK <n>  CLR-MASK <m>  AT-ADDRESS <a> FOR-OUTPUT
DEFINE <name>  AND-MASK <n>  XOR-MASK <m>  AT-ADDRESS <a> FOR-OUTPUT
```

The trinary must be given a name. This name acts like a subroutine: when it is used, the trinary sets and clears bits at the specified address.

The most commonly used output action is SET/CLR. When performed, any "1" bits in the `SET-MASK` will be set at address a. Any "1" bits in the `CLR-MASK` will be *cleared* at address a. You can think of this as lists of bits to be set and bits to be cleared in the register (or memory location). If you need to only set or only clear bits, the unneeeded mask should be zero. For example, to set the LSB at address $F00, you would use

```
DEFINE <name>  SET-MASK 1  CLR-MASK 0  AT-ADDRESS HEX 0F00 FOR-OUTPUT
```

Avoid having the same bit in both the SET-MASK and the CLR-MASK; the result will be indeterminate.[9]

An alternative action is AND/XOR. This can be used to change the state of bits, depending on their current value. When performed, the AND-MASK is applied to the value at address a. Then the XOR-MASK is applied to this result. The final result is stored back to address a. You can thus set, clear, and toggle bits in one operation:

AND-MASK bit	XOR-MASK bit	function
0	0	clears the bit
0	1	sets the bit
1	0	leaves the bit unchanged
1	1	toggles (inverts) the bit

Remember that the AND is always applied before the XOR. Use this form with care: it is very easy to clear bits inadvertently with a badly chosen AND-MASK.

SET-MASK, CLR-MASK, and AT-ADDRESS can be specified in any order. The following are equivalent:

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

Likewise, AND-MASK, XOR-MASK, and AT-ADDRESS can be specified in any order. But you *cannot* mix SET/CLR masks with AND/XOR masks.

Output trinaries can be used in state transitions and in procedural (Forth) code. **You are not required to use trinaries for the compound action in a state transition.** They are merely provided as a convenience.

An output trinary can be moved to Flash ROM with EEWORD:

```
DEFINE <name>  SET-MASK <n>   CLR-MASK <m>  AT-ADDRESS <a> FOR-OUTPUT
EEWORD
```

103. Defining Procedural Actions

For either test conditions or output actions, you may wish to specify procedural code. There is a form of the trinary declaraction that allows this:

```
DEFINE <name>  PROC  ...procedural code...  END-PROC
```

[9] Currently, on the DSP5680x family, these operations are performed by reading memory, applying the logical operations, and then writing the result back to memory. But there is no guarantee that future versions of IsoMax, or versions for other processors, will be implemented in precisely the same way.

When used to specify a test condition, the procedural (Forth) code should leave a true/false value on the stack. When used to specify an output action, the code should expect nothing from the stack, and when finished, leave nothing on the stack.

PROCs can be used within state transitions and in procedural (Forth) code. You are not required to use PROCs; they are provided as a convenience.[10]

These also can be moved to Flash ROM with EEWORD:

```
DEFINE <name>  PROC  ...procedural code...  END-PROC EEWORD
```

104. Activating State Machines

Refer to **Application Note: Starting IsoMax State Machines** for documentation on `SET-STATE`, `INSTALL`, `MACHINE-CHAIN`, and `SCHEDULE-RUNS`.

[10] `DEFINE ... PROC ... END-PROC` simply creates a normal Forth high-level ("colon") definition.

105. IsoMax Performance Monitoring

The IsoMax system is designed to execute user-defined state machines at a regular interval. This interval can be adjusted by the user with the `PERIOD` command. But how quickly can the state machine be executed? IsoMax provides tools to measure this, and also to handle the occasions when the state machine takes "too long" to process.

106. An Example State Machine

For the purposes of illustration, we'll use a state machine that blinks the green LED:[11]

```
LOOPINDEX CYCLE-COUNTER
DECIMAL 100 CYCLE-COUNTER END
1 CYCLE-COUNTER START

MACHINE SLOW_GRN

ON-MACHINE SLOW_GRN
 APPEND-STATE SG_ON
 APPEND-STATE SG_OFF

IN-STATE SG_ON
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED OFF
 THEN-STATE SG_OFF
 TO-HAPPEN

IN-STATE SG_OFF
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED ON
 THEN-STATE SG_ON
 TO-HAPPEN

SG_ON SET-STATE
INSTALL SLOW_GRN
```

This machine will execute at the default rate of `DECIMAL 50000 PERIOD`, or 100 Hz (since the clock rate is 5 MHz).

107. IsoMax Processing Time

Every time IsoMax processes your state machines, it measures the total number number of clock cycles required. This is available to you in three variables:

[11] This example uses `LOOPINDEX` and `INSTALL`, and therefore requires IsoMax v0.36 or later.

`TCFAVG`	This is a moving average of the measured processing time.[12] It is reported as a number of 5 MHz clock cycles.
`TCFMIN`	This is the minimum measured processing time (in 5 MHz cycles). Note that this is *not* automatically reset when you install new state machines. Therefore, after installing new state machines, store a large value in `TCFMIN` to remove the old (false) minimum.
`TCFMAX`	This is the maximum measured processing time (in 5 MHz cycles). This is *not* automatically reset when you change state machines. Therefore, after changing state machines, store a zero in `TCFMAX` to remove the old (false) maximum.

To see this, enter the following commands while the `SLOW_GRN` state machine is running:

```
DECIMAL 50000 TCFMIN !
0 TCFMAX !
TCFAVG ?
TCFMIN ?
TCFMAX ?
```

You may see an AVG and MIN time of about 630 cycles, and a MAX time near 1175 cycles.[13] With a 5 MHz clock, this corresponds to a processing time of about 126 usec (average) and 235 usec (maximum). The average is near the minimum because most of the time, the state machine is performing no action. Only once every 100 iterations does the `CYCLE-COUNTER` expire and force a change of LED state.

`TCFAVG`, `TCFMIN`, and `TCFMAX` return results in the same units used by `PERIOD` (counts of a 5 MHz clock). This means you can use `TCFMAX` to determine the safe lower bound of `PERIOD`. In this case, you could set `PERIOD` as low as 1175 decimal, and IsoMax would always have time to process the state machine.

108. Exceeding the Allotted Time

What if, in this example, `PERIOD` had been set to 1000 decimal? Most of the time, the state machine would be processed in less time, but once per second the LED transition would require more time than was allotted.

IsoMax will handle this gracefully by "skipping" clock interrupts as long as the state machine is still processing. With PERIOD set to 1000, an interrupt occurs every 200 usec. When the LED transition occurs, one interrupt will be skipped, and so there will be 400 usec (2000 cycles) between iterations of the state machine.

[12] To be precise, `TCFAVG` is computed as the arithmetic mean of the latest measurement and the previous average, i.e., Tavg[n+1] = (Tmeasured + Tavg[n]) / 2.

[13] These times were measured on an IsoPod running the v0.37 kernel. With no state machines `INSTALL`ed, the same kernel shows a `TCFAVG` of 88 cycles (17.6 usec). This represents the overhead to respond to a timer interrupt, service it, and perform an empty `INSTALL` list.

If this happens only rarely, it may not be of concern. But if it happens frequently, you may have a problem with your state machine, or you may have set `PERIOD` too low. To let you know when this is happening, IsoMax maintains an "overflow" counter:

`TCFOVFLO` A variable, reset to zero when IsoMax is started, and incremented every time a clock interrupt occurs before IsoMax has completed state processing. (In other words, this tells you the number of "skipped" clock interrupts.)

You can see this in action by typing the following commands while the SLOW_GRN state machine is still running:

```
TCFOVFLO ?
DECIMAL 1000 PERIOD
TCFOVFLO ?
TCFOVFLO ?
TCFOVFLO ?
50000 PERIOD
TCFOVFLO ?
TCFOVFLO ?
```

Be sure to type these commands, and don't just upload them -- you need some time to elapse between commands so that you can see the overflow counter increase. After you change `PERIOD` back to 50000, the overflow counter will stop increasing.

109. Automatic Overflow Processing

If IsoMax overflows happen too frequently, you may wish your application to take some corrective action. You could write a program to monitor the value of `TCFOVFLO`. But IsoMax does this for you, and allows you to set an "alarm" value and an action to be performed:

`TCFALARM` A variable, set to zero when IsoMax is started. If set to a nonzero value, IsoMax will declare an "alarm" condition when the number of timer overflows (`TCFOVFLO`) reaches this value. If set to zero, timer overflows will be counted but otherwise ignored.

`TCFALARMVECTOR` A variable, set to zero when IsoMax is started. If set to a nonzero value, IsoMax will assume that this is the CFA of a Forth word to be executed when an "alarm" condition is declared. This Forth word should be stack-neutral, that is, it should consume no values from the stack, and should leave no values on the stack.

If set to zero, timer overflows will be counted but otherwise ignored.

Note that *both* of these values must be nonzero in order for alarm processing to take place. Be particularly careful that `TCFALARMVECTOR` is set to a valid address; if it is set to an invalid address it is likely to halt the IsoPod.

To continue with the previous example:

```
REDLED OFF
: TOO-FAST   REDLED ON  50000 PERIOD ;
' TOO-FAST CFA  TCFALARMVECTOR !
100 TCFALARM !
0 TCFOVFLO !
```

This defines a word `TOO-FAST` which is to be performed if too many overflows occur. `TOO-FAST` will turn on the red LED, and will also change the IsoMax period to a large (and presumably safe) value. The phrase `' TOO-FAST CFA` returns the Forth CFA of the `TOO-FAST` word; this can be stored as the `TCFALARMVECTOR`. Finally, the alarm threshold is set to 100 overflows, and the overflow counter is reset.[14]

Now watch the LEDs after you type the command

```
1000 PERIOD
```

The slow blinking of the green LED will change to a rapid flicker for a few seconds. Then the red LED will come on and the green LED will return to a slow blink. This was caused by `TOO-FAST` being executed automatically when `TCFOVFLO` reached 100.

110. Counting IsoMax Iterations

It may be necessary for you to know how many times IsoMax has processed the state machine. IsoMax provides another variable to help you determine this:

`TCFTICKS`	A variable, set to zero when IsoMax is started, and incremented on every IsoMax clock interrupt.

The frequency of the IsoMax clock interrupt is set by `PERIOD`; the default value is 100 Hz (50000 cycles of a 5 MHz clock). With this knowledge, you can use `TCFTICKS` for time measurement. With `DECIMAL 50000 PERIOD`, the variable `TCFTICKS` will be incremented 100 times per second.

Note that `TCFTICKS` is incremented *whether or not* an IsoMax overflow occurs. That is, it counts the number of IsoMax clock interrupts, *not* the number of times the state machine was processed. To compute the actual number of executions of the state machine, you must subtract the number of “skipped” clock interrupts, thus:

```
TCFTICKS @ TCFOVFLO @ -
```

[14] The test is for equality (`TCFOVFLO=TCFALARM`), not “greater than,” to ensure that the alarm condition only happens once. The previous exercise left a large value in `TCFOVFLO`; if this is not reset to zero, the alarm won’t occur until `TCFOVFLO` reaches 65535, “wraps around” back to zero, and then counts to 100.

111. Loop Indexes

A LOOPINDEX is an object that counts from a start value to an end value. Its name comes from the fact that it resembles the I index of a DO loop. However, LOOPINDEXes can be used anywhere, not just in DO loops. In particular, they can be used in IsoMax state machines to perform a counting function.

112. Defining a Loop Index

You define a LOOPINDEX just like you define a variable:

```
    LOOPINDEX name
```

...where you choose the "name." For example,

```
    LOOPINDEX CYCLE-COUNTER
```

Once you have defined a LOOPINDEX, you can specify a starting value, an ending value, and an optional step (increment) for the counter. For example, to specify that the counter is to go from 0 to 100 in steps of 2, you would type:

```
    0 CYCLE-COUNTER START
    100 CYCLE-COUNTER END
    2 CYCLE-COUNTER STEP
```

You can specify these in any order. If you don't explicitly specify `START`, `END`, or `STEP`, the default values will be used. The default for a new counter is to count from 0 to 1 with a step of 1. So, if you want to define a counter that goes from 0 to 200 with a step of 1, all you have to change is the `END` value:

```
    LOOPINDEX BLINK-COUNTER
    200 BLINK-COUNTER END
```

If you use a negative `STEP`, the counter will count backwards. In this case the `END` value must be less than the `START` value!

You can change the `START`, `END`, and `STEP` values at any time, even when the counter is running.

113. Counting

The loopindex is incremented when you use the statement

```
    name COUNT
```

For example,

```
    CYCLE-COUNTER COUNT
```

COUNT will always return a truth value which indicates if the loopindex has *passed* its limit. If it has not, COUNT will return false (zero). If it has, COUNT will return true (nonzero), and it will also reset the loopindex value to the START value.

This truth value allows you to take some action when the limit is reached. This can be used in an IF..THEN statement:

```
CYCLE-COUNTER COUNT IF  GRNLED OFF  THEN
```

It can also be used as an IsoMax condition:

```
CONDITION CYCLE-COUNTER COUNT CAUSES GRNLED OFF ...
```

In this latter example, the loopindex will be incremented every time this condition is tested, but the CAUSES clause will be performed only when the loopindex reaches its limit.

Note that the limit test depends on whether STEP is positive or negative. If positive, the loopindex "passes" its limit when the count value + STEP value is *greater than* the END value. If negative, the loopindex passes its limit when the count value + STEP value is *less than* the END value.

In both cases, signed integer comparisons are used. **Be careful** that your loopindex limits don't result in an infinite loop! If you specify an END value of HEX 7FFF, and a STEP of 1, the loopindex will *never* exceed its limit, because in two's complement arithmetic, adding 1 to 7FFF gives -8000 hex -- a negative number, which is clearly *less* than 7FFF.

Also, be careful that you always use or discard the truth value left by COUNT. If you just want to increment the loopindex, without checking if it has passed its limit, you should use the phrase

```
CYCLE-COUNTER COUNT DROP
```

114. Using the Loopindex Value

Sometimes you need to know the value of the index while it is counting. This can be obtained with the statement

```
name VALUE
```

For example,

```
CYCLE-COUNTER VALUE
```

Sometimes you need to manually reset the count to its starting value, before it reaches the end of count. The statement

```
name RESET
```

will reset the index to its `START` value. For example,

```
     CYCLE-COUNTER RESET
```

Remember that you *don't* need to explicitly `RESET` the loopindex when it reaches the end of count. This is done for you automatically. The loopindex "wraps around" to the `START` value, when the `END` value is passed.

115. A "DO loop"Example

This illustrates how a loopindex can be used to replace a DO loop in a program. This also illustrates the use of `VALUE` to get the current value of the loopindex.

```
LOOPINDEX BLINK-COUNTER
DECIMAL 20 BLINK-COUNTER END
2 BLINK-COUNTER STEP
: TEST   BEGIN  BLINK-COUNTER VALUE .  BLINK-COUNTER COUNT
UNTIL ;
```

If you now type `TEST`, you will see the even numbers from 0 (the default `START` value) to 20 (the `END` value).[15] This is useful to show how the loopindex behaves with negative steps:

```
-2 BLINK-COUNTER STEP
40 BLINK-COUNTER START
BLINK-COUNTER RESET
TEST
```

This counts backwards by twos from 40 to 20. Note that, because we changed the `START` value of `BLINK-COUNTER`, we had to manually `RESET` it. Otherwise `TEST` would have started with the index value left by the previous `TEST` (zero), and it would have immediately terminated the loop (because it's less than the `END` value of 20).

116. An IsoMax Example

This example shows how a loopindex can be used within an IsoMax state machine, and also illustrates one technique to "slow down" the state transitions. Here we wish to blink the green LED at a rate 1/100 of the normal state processing speed. (Recall that IsoMax normally operates at 100 Hz; if we were to blink the LED at this rate, it would not be visible!)

```
LOOPINDEX CYCLE-COUNTER
DECIMAL 100 CYCLE-COUNTER END
1 CYCLE-COUNTER START

MACHINE SLOW_GRN
```

[15] Forth programmers should note that the LOOPINDEX continues *up to and including* the `END` value, whereas a comparable DO loop continues only *up to* (but not including) its limit value.

```
ON-MACHINE SLOW_GRN
 APPEND-STATE SG_ON
 APPEND-STATE SG_OFF

IN-STATE SG_ON
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED OFF
 THEN-STATE SG_OFF
 TO-HAPPEN

IN-STATE SG_OFF
 CONDITION CYCLE-COUNTER COUNT
 CAUSES GRNLED ON
 THEN-STATE SG_ON
 TO-HAPPEN

SG_ON SET-STATE
INSTALL SLOW_GRN
```

Here the loopindex `CYCLE-COUNTER` counts from 1 to 100 in steps of 1. It counts in *either* state, and only when the count reaches its limit do we change to the other state (and change the LED). That is, the end-of-count `CAUSES` the LED action and the change of state. Since the counter is automatically reset after the end-of-count, we don't need to explicitly reset it in the IsoMax code.

117. Summary of Loopindex Operations

`LOOPINDEX name`

Defines a "loop index" variable with the given `name`. For example,
```
   LOOPINDEX COUNTER1
```

`START`
`END`
`STEP`

These words set the start value, the end value, or the step value (increment) for the given loop index. All of these expect an integer argument and the name of a loopindex variable. Examples:
```
     1 COUNTER1 START
   100 COUNTER1 END
     3 COUNTER1 STEP
```
These can be specified in any order. If any of them is not specified, the default values will be used (START=0, END=1, STEP=1).

`COUNT`

This causes the given loop index to increment by the STEP value, and returns a true or false value: true (-1) if the end of count was reached, false (0) otherwise. For example:
```
   COUNTER1 COUNT
```
End of count is determined after the loop index is incremented, as follows: If STEP is positive, "end of count" is when the index is greater than the END value. If STEP is negative, "end of count" is when the index is less than the END value. Signed integer comparisons are used. In either case, when the end of count is reached, the loop index is reset to its START value.

`RESET`

This word manually resets the given loop index to its START value. Example:
```
   COUNTER1 RESET
```

`VALUE`

This returns the current index value (counter value) of the given loop index. It will return a signed integer in the range -32768..+32767. For example:
```
   COUNTER1 VALUE .
```
...prints the loop index COUNTER1

118. Autostarting an IsoMax Application

119. The Autostart Search

When the IsoPod is reset, it searches the Program Flash ROM for an **autostart pattern**. This is a special pattern in memory which identifies an autostart routine. It consists of the value $A55A, followed by the address of the routine to be executed.

```
xx00:  $A55A
xx01:  address of routine
```

It must reside on an address within Program ROM which is a multiple of $400, i.e., $0400, $0800, $0C00, ... $7400, $7800, $7C00.

The search proceeds from $0400 to $7C00, and terminates when the *first* autostart pattern is found. This routine is then executed. If the routine exits, the IsoMax interpreter will then be started.

120. Writing an Application to be Autostarted

Any defined word can be installed as an autostart routine. For embedded applications, this routine will probably be an endless loop that never returns.

Here's a simple routine that reads characters from terminal input, and outputs their hex equivalent:

```
: MAIN   HEX BEGIN KEY . AGAIN ;   EEWORD
```

Note the use of `EEWORD` to put this routine into Flash ROM. An autostart routine must reside in Flash ROM, because when the IsoPod is powered off, the contents of RAM will be lost. If you install a routine in Program RAM as the autostart routine, the IsoPod will crash when you power it on. (To recover from such a crash, see "Bypassing the Autostart" below.)

Because this definition of `MAIN` uses a `BEGIN...AGAIN` loop, it will run forever. You can define this word from the keyboard and then type `MAIN` to try it out (but you'll have to reset the IsoPod to get back to the command interpreter). This is how you would write an application that is to run forever when the IsoPod is reset.

You can also write an autostart routine that exits after performing some action. One common example is a routine that starts some IsoMax state machines. For this discussion, we'll use a version of `MAIN` that returns when an escape character is input:

```
HEX
: MAIN2   HEX BEGIN KEY DUP .  1B = UNTIL ;   EEWORD
```

In this example the loop will run continuously until the ESC character is received, then it exits normally. If this is installed as the autostart routine, when it exits, the IsoPod will proceed to start the IsoMax command interpreter.

121. Installing an Autostart Application

One the autostart routine is written, it can be installed into Flash ROM with the command

```
    address AUTOSTART routine-name
```

This will build the autostart pattern in ROM. The *address* is the location in Flash ROM to use for the pattern, and must be a multiple of $400. Often the address $7C00 is used. This leaves the largest amount of Flash ROM for the application program, and leaves the option of later programming a new autostart pattern at a lower address. (Remember, the autostart search starts low and works up until the *first* pattern found, so an autostart at $7800 will override an autostart at $7C00.) So, for example, you could use

```
    HEX 7C00 AUTOSTART MAIN2
```

to cause the word `MAIN2` to be autostarted. (Note the use of the word `HEX` to input a hex number.)

Try this now, and then reset the IsoPod. You'll see that no "IsoMax" prompt is displayed. If you start typing characters at the terminal, you'll see the hex equivalents displayed. This will continue forever until you hit the ESC key, at which point the "IsoMax" prompt is displayed and the IsoPod will accept commands.

122. Saving the RAM data for Autostart

Power the IsoPod off, and back on, and observe that the autostart routine still works. Then press the ESC key to exit to the IsoMax command interpreter. Now try typing `MAIN2`. IsoMax doesn't recognize the word, even though you programmed it into Flash ROM! If you type `WORDS` you won't see `MAIN2` in the listing. Why?

The reason is that some information about the words you have defined is kept in RAM[16]. If you just reset the board from MaxTerm, the RAM contents will be preserved. But if you power the board off and back on, the RAM contents will be lost, and IsoMax will reset RAM to known defaults. If you type `WORDS` after a power cycle, all you will see are the standard IsoMax words: all of your user-defined words are lost.

To prevent this from happening, you must save the RAM data to be restored on reset. This is done with the word `SAVE-RAM`:

```
    SAVE-RAM
```

[16] To be specific, what is lost is the LATEST pointer, which always points to the last-defined word in the dictionary linked list. The power-up default for this is the last-defined word in the IsoMax kernel.

This can be done either just before, or just after, you use AUTOSTART. SAVE-RAM takes a "snapshot" of the RAM contents, and stores it in Data Flash ROM. Then, the next time you power-cycle the board, those preserved contents will be reloaded into RAM. This includes *both* the IsoMax system variables, and any variables or data structures you have defined.

Note: a simple reset will not reload the RAM. When the IsoPod is reset, it first checks to see if it has lost its RAM data. Only if the RAM has been corrupted -- as it is by a power loss -- will the IsoPod attempt to load the SAVE-RAM snapshot. (And only if there is no SAVE-RAM snapshot will it restore the factory defaults.) If you use MaxTerm to reset the IsoPod, the RAM contents will be preserved.

123. Removing an Autostart Application

Don't try to reprogram MAIN2 just yet. Even though the RAM has been reset to factory defaults, MAIN2 is still programmed into Flash ROM, and IsoMax doesn't know about it. In fact, if you try to redefine MAIN2 at this point, you might crash the IsoPod, as it attempts to re-use Flash ROM which hasn't been erased. (To recover from this, see "Bypassing the Autostart," below.)

To completely remove all traces of your previous work, use the word SCRUB:

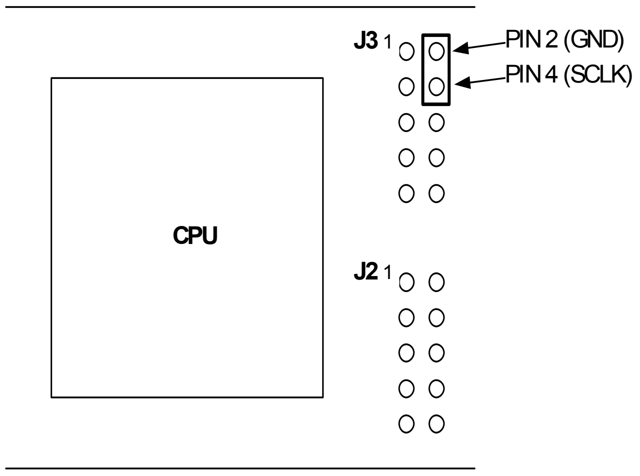
```
SCRUB
```

This will erase all of your definitions from Program Flash ROM -- including any AUTOSTART patterns which have been stored -- and will also erase any SAVE-RAM snapshot from Data Flash ROM. Basically, the word SCRUB restores the IsoPod to its factory-fresh state.

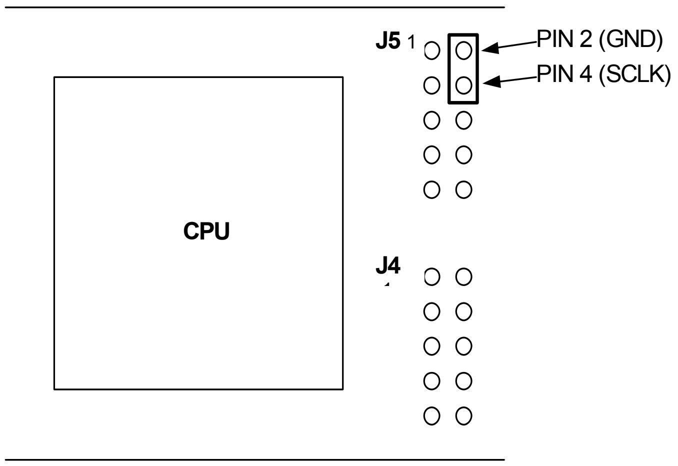
124. Bypassing the Autostart

What if your autostart routine locks up? If you can't get access to the IsoMax command interpreter, how do you SCRUB the application and restore the IsoPod to usability?

You can bypass the autostart search, and go directly to the IsoMax interpreter, by jumpering together pins 2 and 4 on connector J3, and then resetting the IsoPod. You can do this with a common jumper block:



IsoPod V1



IsoPod V2

This connects the SCLK/PE4 pin to ground. When the IsoPod detects this condition on reset, it does not perform the autostart search.

Note that this does *not* erase your autostart application or your `SAVE-RAM` snapshot from Flash ROM. These are still available for your inspection[17]. If you remove the jumper block and reset the IsoPod, it will again try to run your autostart application. (This can be a useful field diagnostic tool.)

To remove your application and start over, you'll need to use the `SCRUB` command. The steps are as follows:

[17] The IsoPod RAM will be reset to factory defaults instead of to the saved values, but you can still examine the `SAVE-RAM` snapshot in Flash ROM.

1. Connect a terminal (or MaxTerm) to the RS-232 port.

2. Jumper pins 2 and 4 on J3.

3. Reset the IsoPod. You will see the "IsoMax" prompt.

4. Type the command `SCRUB` .

5. You can now remove the jumper from J3.

125. Summary

Use `EEWORD` to ensure that all of your application routines are in Flash ROM.

When your application is completely loaded, use `SAVE-RAM` to preserve your RAM data in Flash ROM.

Use `address AUTOSTART routine-name` to install your routine for autostarting. "address" must be a multiple of $0400 in empty Flash ROM; `HEX 7C00` is commonly used.

To clear your application and remove the autostart, use `SCRUB`. This restores the IsoPod to its factory-new state.

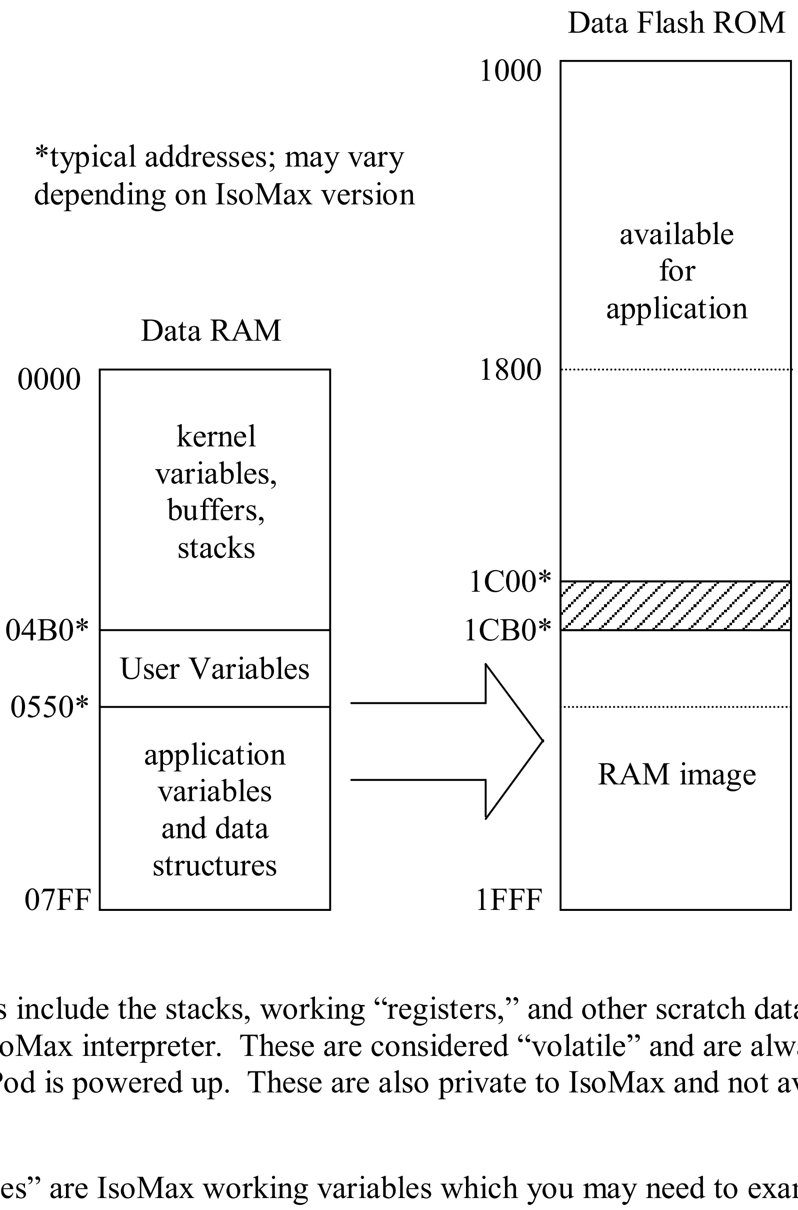
If the autostart application locks up, jumper together pins 2 and 4 of J3, and reset the IsoPod. This will give you access to the IsoMax command interpreter.

126. SAVE-RAM

The IsoPod contains 4K words of nonvolatile "Flash" data storage. This can be used to save system variables and your application variables so that they are automatically initialized when the IsoPod is powered up. This is done with the word `SAVE-RAM`.

127. Data Memory Map

The internal RAM of the IsoPod is divided into three regions: kernel buffers, User Variables, and application variables.


Data Flash ROM
1000
*typical addresses; may vary depending on IsoMax version
available for application
Data RAM
0000
kernel variables, buffers, stacks
1800
1C00*
04B0*
1CB0*
User Variables
0550*
application variables and data structures
RAM image
07FF
1FFF

Kernel buffers include the stacks, working "registers," and other scratch data that are used by the IsoMax interpreter. These are considered "volatile" and are always cleared when the IsoPod is powered up. These are also private to IsoMax and not available to you.

"User Variables" are IsoMax working variables which you may need to examine or change. These include such values as the current number base (`BASE`), the current ROM and RAM allocation pointers, and the Terminal Input Buffer. This region also includes RAM for the IsoMax state machine and the predefined IsoPod I/O objects.

Application data is whatever variables, objects, and buffers you define in your application program. This can extend up to the end of RAM (address 07FF hex in the IsoPod).

128. Saving the RAM image

The word `SAVE-RAM` copies the User Variables and application data to the *end* of Data Flash ROM. All of internal RAM, starting at the first User Variable (currently `C/L`) and continuing to the end of RAM, is copied to corresponding addresses in the Flash ROM.

Note that this will copy all `VARIABLE`s and the RAM contents of all objects, but it will *not* copy the stacks.

Normally you will use `SAVE-RAM` to take a "snapshot"of your RAM data when all your variables are initialized and your application is ready to run.

129. Flash erasure

Because the `SAVE-RAM` uses Flash memory, it must erase the Flash ROM before it can copy to it. This is automatically done by `SAVE-RAM`, and you need not perform any explicit erase function. However, you should be aware that `SAVE-RAM` will erase more Flash ROM than is needed for the RAM image.

Flash ROM is erased in "pages" of 256 words each. To ensure that all of the RAM image is erased, `SAVE-RAM` must erase starting at the next lower page boundary. A page boundary address is always of the form $XX00 (the low eight bits are zero). So, in the illustrated example, Flash ROM is erased starting at address $1C00.

If you use Data Flash ROM directly in your application, you can be sure that your data will be safe if you restrict your usage to addresses $1000-$17FF. Some of the space above $1800 is currently unused, but this is not guaranteed for future IsoMax releases.

130. Restoring the RAM image

The IsoPod will automatically copy the saved RAM image from Flash ROM back to RAM when it is first powered up. This will occur before your application program is started. So, you can use `SAVE-RAM` to create an "initial RAM state" for your application.

If the IsoPod is reset and the RAM contents appear to be valid, the saved RAM image will *not* be used. This may happen if the IsoPod receives a hardware reset signal while power is maintained. Usually this is the desired behavior.

131. Restoring the RAM image manually

You can force RAM to be copied from the saved image by using `RESTORE-RAM`. This does exactly the reverse of `SAVE-RAM`: it copies the contents of Data Flash ROM to Data RAM. The address range copied is the same as used by `SAVE-RAM`.

So, if your application needs RAM to be initialized on every hardware reset (and not just on a power failure), you can put `RESTORE-RAM` at the beginning of your autostart routine.

Note: do not use `RESTORE-RAM` *if* `SAVE-RAM` *has not been performed.* This will cause invalid data to be written to the User Variables (and to your application variables as well), which will almost certainly crash the IsoPod. For most applications it is sufficient, and safer, to use the default RAM restore which is built into the IsoPod kernel.

132. 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[18], 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.

133. 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.

[18] 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.

134. 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.

135. 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
```

136. 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`

137. 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.*

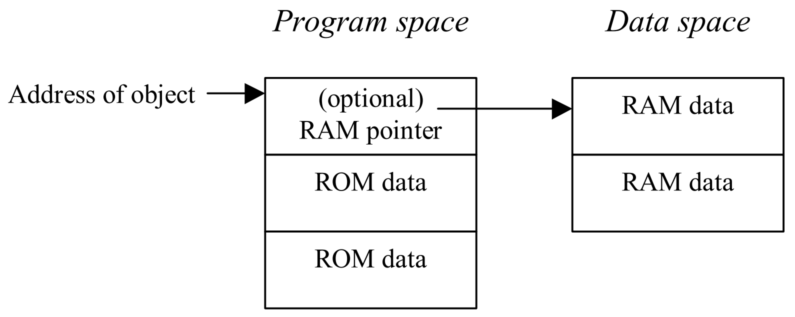
138. Referencing an object

`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.*

139. 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.

140. 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`.

141. Machine Code Programming

IsoMax allows individual words to be written in machine code as well as “high-level” language code. Such words are indistinguishable in function from high-level words, and may be used freely in application programs and state machines.

142. Assembler Programming

The IsoPod uses the Motorola DSP56F805 microprocessor. The machine language of this processor is described in Motorola's *DSP56800 16-Bit Digital Signal Processor Family Manual,* available at

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

IsoMax does *not* include a symbolic assembler for this processor. You must use an external assembler to convert your program to the equivalent hexadecimal machine code, and then insert these numeric opcodes and operands into your IsoMax source code.[19] For an example, let's use an assembler routine to stop Timer C2:

```
; Timer/Counter
; -------------
; Timer control register
;  000x xxxx xxxx xxxx = no count
andc    #$1FFF,X:$0D56  ; TMRC2_CTRL

; Timer status & control register
; Clear TCF flag, clear interrupt enable flag
bfclr   #$8000,X:$0D57  ; TMRC2_SCR  clear TCF
bfclr   #$4000,X:$0D57  ; TMRC2_SCR  clear TCFIE
```

Translated to machine code, this is:

```
80F4    andc    #$1FFF,X:$0D56
0D56
E000
80F4    bfclr   #$8000,X:$0D57
0D57
8000
80F4    bfclr   #$4000,X:$0D57
0D57
4000
```

[19] If you wish to translate your programs manually to machine code, a summary chart of DSP56800 instruction encoding is given at the end of this application note.

To compile this manually into an IsoMax word, you must append each hexadecimal value to the dictionary with the `P,` operator. (The “P” refers to Program space,where all machine code must reside.) You can put more than one value per line:

```
80F4 P, 0D56 P, E000 P,
80F4 P, 0D57 P, 8000 P,
80F4 P, 0D57 P, 4000 P,
```

All that remains is to add this as a word to the IsoMax dictionary, and to return from the assembler code to IsoMax. There are three ways to do this: with `CODE`, `CODE-SUB`, and `CODE-INT`.

143. CODE functions

The special word `CODE` defines a machine language word as follows:

```
CODE word-name
```

(machine language for your word)

(machine language for `JMP NEXT`)

```
END-CODE
```

Machine code words that are created with `CODE` must return to IsoMax by performing a jump to the special address `NEXT`. *In IsoMax versions 0.52 and higher, this is address $0080. Earlier versions of IsoMax do not support NEXT and you must use CODE-SUB, described below, to write machine code words.*

An absolute jump instruction is $E984. Thus a `JMP NEXT` translates to $E984 $0080, and our example `STOP-TIMERC2` word could be written as follows:

```
HEX
CODE STOP-TIMERC2
    80F4 P, 0D56 P, E000 P,
    80F4 P, 0D57 P, 8000 P,
    80F4 P, 0D57 P, 4000 P,
    E984 P, 0080 P, ( JMP NEXT )
END-CODE
```

Remember, this example will only work on recent versions of IsoMax (0.52 or later).

144. CODE-SUB functions

The special word `CODE-SUB` is just like `CODE`, except that the machine code returns to IsoMax with an ordinary `RTS` instruction. This can be useful if you need to write a machine code routine that can be called both from IsoMax and from other machine code routines. It's also useful if the `NEXT` address is not available (as in IsoMax versions prior to 0.52). The syntax is similar to `CODE`:

```
CODE-SUB word-name
```

(machine language for your word)

(machine language for `RTS`)

```
END-CODE
```

An `RTS` instruction is $EDD8, so `STOP-TIMERC2` could be written with `CODE-SUB` as follows:

```
HEX
CODE-SUB STOP-TIMERC2
    80F4 P, 0D56 P, E000 P,
    80F4 P, 0D57 P, 8000 P,
    80F4 P, 0D57 P, 4000 P,
    EDD8 P, ( RTS )
END-CODE
```

This example will work in all versions of IsoMax.

145. CODE-INT functions

`CODE-INT` is just like `CODE-SUB`, except that the machine code returns to IsoMax with an `RTI` (Return from Interrupt) instruction, $EDD9. This is useful if you need to write a machine code interrupt handler that can also be called directly from IsoMax. *CODE-INT is only available on IsoMax versions 0.52 and later.*

```
HEX
CODE-INT STOP-TIMERC2
    80F4 P, 0D56 P, E000 P,
    80F4 P, 0D57 P, 8000 P,
    80F4 P, 0D57 P, 4000 P,
    EDD9 P, ( RTI )
END-CODE
```

To obtain the address of the machine code after it is compiled, use the phrase

```
      ' word-name CFA 2+
```

Note: if you are using EEWORD to put this new word into Flash ROM, use EEWORD *before* trying to obtain the address of the machine code. EEWORD will change this address.

146. Register Usage

In the current version of IsoMax software, all DSP56800 address and data registers may be used in your `CODE` and `CODE-SUB` words. You need not preserve R0-R3, X0, Y0, Y1, A, B, or N. Do not change the "mode" registers M01 or OMR, and do not change the stack pointer SP.

Future versions of IsoMax may add more restrictions on register use. If you are concerned about compatibility with future kernels, you should save and restore all registers that your machine code will use.

`CODE-INT` words are expected to be called from interrupts, and so they should save any registers that they use.

147. Calling High-Level Words from Machine Code

You can call a high-level IsoMax word from within a machine-code subroutine. This is done by calling the special subroutine `ATO4` with the address of the word you want to execute.[20] This address must be a Code Field Address (CFA) and is obtained with the phrase

```
     ' word-name CFA
```

This address must be passed in register R0. You can load a value into R0 with the machine instruction $87D0, $xxxx (where xxxx is the value to be loaded).

The address of the `ATO4` routine can be obtained from a constant named `ATO4`. You can use this constant directly when building machine code. The opcode for a JSR instruction is $E9C8, $aaaa where aaaa is an absolute address. So, to write a `CODE-SUB` routine that calls the IsoMax word `DUP`, you could write:

```
HEX
CODE-SUB NEWDUP
    87D0 P, ' DUP CFA P, ( move DUP CFA to R0 )
    E9C8 P, ATO4 P,      ( JSR ATO4 )
    EDD8 P,         ( RTS )
END-CODE
```

Observe that the phrases `' DUP CFA` and `ATO4` are used *within* the CODE-SUB to generate the proper addresses where required.

[20]The name ATO4 comes from "Assembler to Forth" and refers to the Forth underpinnings of IsoMax.

148. Using CPU Interrupts in the IsoPod

This applies to IsoPod kernel v0.38 and later.

149. Interrupt Vectors in Flash ROM

The DSP56F805 processor used in the IsoPod supports 64 interrupt vectors, in the first 128 locations of Flash ROM. Each vector is a two-word machine instruction, normally a JMP instruction to the corresponding interrupt routine. When an interrupt occurs, the CPU jumps directly to the appropriate address ($00-$7E) in the vector table.

Since this vector table is part of the IsoPod kernel, it cannot be altered by the user. Also, some interrupts are required for the proper functioning of the IsoPod, and these vectors must never be changed. So the IsoPod includes a "user" vector table at the high end of Flash ROM (addresses $7D80-7DFE). This is exactly the same as the "kernel" vector table, except that certain "reserved for IsoPod" interrupts have been excluded. The user vector table can be programmed, erased, and reprogrammed freely by the user, as long as suitable precautions are taken.

150. Writing Interrupt Service Routines

Interrupt service routines must be written in DSP56F805 machine language, and must end with an RTI (Return from Interrupt) instruction. Some peripherals will have additional requirements; for example, many interrupt sources need to be explicitly cleared by the interrupt service routine. For more information about interrupt service routines, refer to the Motorola DSP56800 16-Bit Digital Signal Processor Family Manual (Chapter 7), and the Motorola DSP56F801/803/805/807 16-Bit Digital Signal Processor User's Manual.

You should be aware that the IsoPod uses certain channels in the Interrupt Priority controller:

> The IsoMax Timer (Timer D3) is assigned to Interrupt Priority Channel 3.
> SCI#0 (RS-232) serial I/O is assigned to Interrupt Priority Channel 4.
> The I/O Scheduling Timer[21] is assigned to Interrupt Priority Channel 5.

These channels may be shared by other peripherals. However, it is important to remember that these channels are *enabled* by the IsoMax kernel after a reset, and must never be disabled. You should not use the corresponding bits in the Interrupt Priority Register as interrupt enable/disable bits.

Interrupt channels 0, 1, 2, and 6 are reserved for your use. The IsoMax kernel does not use them, and you may assign, enable, or disable them freely. Channel 0 has the lowest priority, and 6 the highest.[22]

[21] This will be a feature of future IsoMax kernels. Interrupt channel 5 is reserved for this use.

151. The User Interrupt Vector Table

The user vector table is identical to the kernel (CPU) vector table, except that it starts at address $7D80 instead of address $0. Each interrupt vector is two words in this table, sufficient for a machine language jump instruction. For all interrupts which are not reserved by IsoMax, the kernel vector table simply jumps to the corresponding location in the user vector table. (Remember that this adds the overhead of one absolute jump instruction -- 6 machine clock cycles -- to the interrupt service.)

Note: IsoPod kernels version 0.37 and earlier do not support a user vector table.

Note: This table is subject to change. Future versions of the IsoPod software may reserve more of these interrupts for internal use, as more I/O functions are added to the IsoPod kernel.

Interrupt Number	User Vector Address	Kernel Vector Address	Description
0		$00	reset - *reserved for IsoPod*
1	$7D82	$02	COP Watchdog reset
2	$7D84	$04	reserved by Motorola
3		$06	illegal instruction - *reserved for IsoPod*
4	$7D88	$08	Software interrupt
5	$7D8A	$0A	hardware stack overflow
6	$7D8C	$0C	OnCE Trap
7	$7D8E	$0E	reserved by Motorola
8	$7D90	$10	external interrupt A
9	$7D92	$12	external interrupt B
10	$7D94	$14	reserved by Motorola
11	$7D96	$16	boot flash interface
12	$7D98	$18	program flash interface
13	$7D9A	$1A	data flash interface
14	$7D9C	$1C	MSCAN transmitter ready
15	$7D9E	$1E	MSCAN receiver full
16	$7DA0	$20	MSCAN error
17	$7DA2	$22	MSCAN wakeup
18	$7DA4	$24	reserved by Motorola
19	$7DA6	$26	GPIO E
20	$7DA8	$28	GPIO D
21	$7DAA	$2A	reserved by Motorola
22	$7DAC	$2C	GPIO B
23	$7DAE	$2E	GPIO A
24	$7DB0	$30	SPI transmitter empty
25	$7DB2	$32	SPI receiver full/error
26	$7DB4	$34	Quad decoder #1 home
27	$7DB6	$36	Quad decoder #1 index pulse
28	$7DB8	$38	Quad decoder #0 home
29	$7DBA	$3A	Quad decoder #0 index pulse

[22] Use channel 6 only for critically-urgent interrupts, since it will take priority over channels 4 and 5, both of which require prompt service.

Interrupt Number	User Vector Address	Kernel Vector Address	Description
30	$7DBC	$3C	Timer D Channel 0
31	$7DBE	$3E	Timer D Channel 1
32	$7DC0	$40	Timer D Channel 2
33		$42	Timer D Channel 3 - *reserved for IsoPod*
34	$7DC4	$44	Timer C Channel 0
35	$7DC6	$46	Timer C Channel 1
36	$7DC8	$48	Timer C Channel 2
37	$7DCA	$4A	Timer C Channel 3
38	$7DCC	$4C	Timer B Channel 0
39	$7DCE	$4E	Timer B Channel 1
40	$7DD0	$50	Timer B Channel 2
41	$7DD2	$52	Timer B Channel 3
42	$7DD4	$54	Timer A Channel 0
43	$7DD6	$56	Timer A Channel 1
44	$7DD8	$58	Timer A Channel 2
45	$7DDA	$5A	Timer A Channel 3
46	$7DDC	$5C	SCI #1 Transmit complete
47	$7DDE	$5E	SCI #1 transmitter ready
48	$7DE0	$60	SCI #1 receiver error
49	$7DE2	$62	SCI #1 receiver full
50	$7DE4	$64	SCI #0 Transmit complete
51		$66	SCI #0 transmitter ready - *reserved for IsoPod*
52	$7DE8	$68	SCI #0 receiver error
53		$6A	SCI #0 receiver full - *reserved for IsoPod*
54	$7DEC	$6C	reserved by Motorola
55	$7DEE	$6E	ADC A Conversion complete
56	$7DF0	$70	reserved by Motorola
57	$7DF2	$72	ADC A zero crossing/error
58	$7DF4	$74	Reload PWM B
59	$7DF6	$76	Reload PWM A
60	$7DF8	$78	PWM B Fault
61	$7DFA	$7A	PWM A Fault
62	$7DFC	$7C	PLL loss of lock
63	$7DFE	$7E	low voltage detector

152. Clearing the User Vector Table

Since the user vector table is at the high end of Flash ROM, it will be erased by the SCRUB command (which erases all of the user-programmable Flash ROM).

If you wish to erase *only* the user vector table, you should use the command

```
    HEX 7D00 PFERASE
```

This will erase 256 words of Program Flash ROM, starting at address 7D00. In other words, this will erase locations 7D00-7DFF, which includes the user vector table. Because of the limitations of Flash ROM, you cannot erase a smaller segment -- you must erase 256 words. However, this is at the high end of Flash ROM and is unlikely to affect your application program, which is built upward from low memory.

When Flash ROM is erased, all locations read as $FFFF. This is an illegal CPU instruction. So it is very important that you install an interrupt vector *before* you enable the corresponding interrupt! If you enable a peripheral interrupt when no vector has installed, you will cause an Illegal Instruction trap and the IsoPod will reset.[23]

153. Installing an Interrupt Vector

Once the Flash ROM has been erased, you can write data to it with the `PF!` operator. Each location can be written only once, and must be erased before being written with a different value.[24]

For example, this will program the low-voltage-detect interrupt to jump to address zero. (This will restart the IsoPod, since address zero is the reset address.)

```
      HEX  E984 7DFE PF!  0 7DFF PF!
```

E984 is the machine language opcode for an absolute jump; this is written into the first word of the vector. The destination address, 0, is written into the second word. Because these addresses are in Flash ROM, you must use the `PF!` operator. An ordinary `!` operator will not work.

154. Precautions when using Interrupts

1. An unprogrammed interrupt vector will contain an FFFF instruction, which is an illegal instruction on the DSP56F805. Don't enable an interrupt until *after* you have installed its interrupt vector.

2. Remember that most interrupts must be cleared at the source before your service routine Returns from Interrupt (with an RTI instruction). If you forget to clear the interrupt, you may end in an infinite loop.

3. Remember that `SCRUB` will erase all vectors in the user table. Be sure to disable *all* of the interrupts that you have enabled, before you use `SCRUB`.

4. You cannot erase a single vector in the user table. You must use `HEX 7D00 PFERASE` to erase the entire table. As with `SCRUB`, be sure to disable all of your interrupt sources first.

5. Do *not* use the global interrupt enable (bits I1 and I0 in the Status Register) to disable your peripheral interrupts. This will also shut off the interrupts that are used by IsoMax, and the IsoPod will likely halt.

6. It *is* permissible to disable interrupts globally for extremely brief periods -- on the order of a few machine instructions -- in order to perform operations that mustn't be

[23] This is why the "illegal instruction" interrupt is reserved for IsoMax. If it were vectored to the user table, and you did not install a vector for it, the attempt to service an illegal instruction would cause yet another illegal instruction, and the CPU would lock up.
[24] Strictly speaking, you can write a Flash ROM location more than once, but you can only change "1" bits to "0." Once a bit has been written as "0", you need to erase the ROM page to return it to a "1" state.

interrupted. But this may affect critical timing within IsoMax, and is generally discouraged.

7. You can perform the action of an IsoPod reset by jumping to absolute address zero. But note that, unlike a true hardware reset, this will *not* disable any interrupt sources that you may have enabled.

155. Interrupt Handlers in High-Level Code

Interrupt handlers must be written in machine code. However, you can write a machine code “wrapper” that will call a high-level IsoMax word to service an interrupt. This application note describes how. You may find it useful to refer to the application notes *Machine Code Programming* and *Using CPU Interrupts in the IsoPod*.

156. How it Works

The machine code routine below works by saving all the registers used by IsoMax, and then calling the ATO4 routine to run a high-level IsoMax word. The high-level word returns to the machine code, which restores registers and returns from the interrupt.

```
HEX 0041 CONSTANT WP

CODE-SUB INT-SERVICE
DE0B P,          \  LEA  (SP)+
D00B P,          \  MOVE X0,X:(SP)+
D10B P,          \  MOVE Y0,X:(SP)+
D30B P,          \  MOVE Y1,X:(SP)+
D08B P,          \  MOVE A0,X:(SP)+
D60B P,          \  MOVE A1,X:(SP)+
D28B P,          \  MOVE A2,X:(SP)+
D18B P,          \  MOVE B0,X:(SP)+
D70B P,          \  MOVE B1,X:(SP)+
D38B P,          \  MOVE B2,X:(SP)+
D80B P,          \  MOVE R0,X:(SP)+
D90B P,          \  MOVE R1,X:(SP)+
DA0B P,          \  MOVE R2,X:(SP)+
DB0B P,          \  MOVE R3,X:(SP)+
DD0B P,          \  MOVE N,X:(SP)+
DE8B P,          \  MOVE LC,X:(SP)+
DF8B P,          \  MOVE LA,X:(SP)+
F854 P, OBJREF P, \  MOVE X:OBJREF,R0
FA54 P, WP P,    \  MOVE X:WP,R2
D80B P,          \  MOVE R0,X:(SP)+
DA1F P,          \  MOVE R2,X:(SP)   ; Note no increment on
last push!
87D0 P, xxxx P,  \  MOVE #$XXXX,R0   ; This is the CFA of
the word to execute
E9C8 P, ATO4 P,  \  JSR  ATO4        ; do that Forth word
FA1B P,          \  MOVE X:(SP)-,R2  ; restore the saved wp
F81B P,          \  MOVE X:(SP)-,R0  ; restore the saved
objref
FF9B P,          \  MOVE X:(SP)-,LA
DA54 P, WP P,    \  MOVE R2,X:FWP
D854 P, OBJREF P, \  MOVE R0,X:OBJREF
FE9B P,          \  MOVE X:(SP)-,LC
FD1B P,          \  MOVE X:(SP)-,N
```

```
FB1B P,          \  MOVE X:(SP)-,R3
FA1B P,          \  MOVE X:(SP)-,R2
F91B P,          \  MOVE X:(SP)-,R1
F81B P,          \  MOVE X:(SP)-,R0
F39B P,          \  MOVE X:(SP)-,B2
F71B P,          \  MOVE X:(SP)-,B1
F19B P,          \  MOVE X:(SP)-,B0
F29B P,          \  MOVE X:(SP)-,A2
F61B P,          \  MOVE X:(SP)-,A1
F09B P,          \  MOVE X:(SP)-,A0
F31B P,          \  MOVE X:(SP)-,Y1
F11B P,          \  MOVE X:(SP)-,Y0
F01B P,          \  MOVE X:(SP)-,X0
EDD9 P,          \  RTI
END-CODE
```

The only registers that are saved automatically by the processor are PC and SR. *All* other registers that will be used must be saved manually. To allow a high-level routine to execute, we must save R0-R3, X0, Y0, Y1, A, B, N, LC, and LA. Two registers that need not be saved are M01 and OMR, because these registers are never used or changed by IsoMax. We must also save the two variables `WP` and `OBJREF`, which are used by the IsoMax interpreter and object processor.

Since the DSP56F805 processor does not have a "pre-increment" address mode, the first push must be *preceded* by a stack pointer increment, LEA (SP)+, and the last push must *not* increment SP.

The instruction ordering may seem peculiar; this is because a MOVE to an address reigster (Rn) has a one-instruction delay. So we always interleave another unrelated instruction after a MOVE x, Rn. Note also the use of the symbols `ATO4` and `OBJREF` to obtain addresses. The variable `WP` is located at hex address 0041 in current IsoMax kernels, and this is defined as a constant for readability.

The value shown as "xxxx" in the listing above is where you must put the Code Field Address (CFA) of the desired high-level word. You can obtain this address with the phrase

```
    ' word-name CFA
```

157. Use of Stacks

The interrupt routine will use the same Data and Return stacks as the IsoMax command interpreter, that is, the "main" program.[25] Normally this is not a problem, because pushing new data onto a stack does not affect the data which is already there. However, you must take care that your interrupt handler leaves the stacks as it found them – that is, does not leave any extra items on the stack, or consume any items that were already there. A stack imbalance in an interrupt handler is a very quick way to crash the IsoPod.

[25]The IsoMax state machine uses an independent set of stacks.

158. Use of Variables

Some high-level words use temporary variables and buffers which are not saved when an interrupt occus. One example is the numeric output functions (`.` `D.` `F.` and the like). You should not use these words within your interrupt routine, since this will corrupt the variables that might be used by the main program.

159. Re-Entrancy

To avoid re-entrancy problems, it is best to *not* re-enable interrupts within your high-level interrupt routine. Interrupts will be re-enabled automatically by the `RTI` instruction, when your routine has finished its processing.

You must of course be sure to clear the interrupt source in your high-level service routine. If you fail to do so, when the `RTI` instruction is executed, a new interrupt will instantly occur, and your program will be stuck in an infinite loop of interrupts.

160. Example: Millisecond Timer

This example uses Timer C2 to increment a variable at a rate of once per millisecond. After loading the entire example, you can use `START-TMRC2` to initialize the timer, set up the interrupt controller for that timer, and enable the interrupt. From that point on, the variable `TICKS` will be incremented on every interrupt. You can fetch the `TICKS` variable in your main program (or from the command interpreter).

The high-level interrupt service routine is `INT-SERVICE`. It does only two things. First it clears the interrupt source, by clearing the TCF bit in the Timer C2 Status and Control Register. Then it increments the variable `TICKS`. As a rule, interrupt service routines should be as short and simple as possible. Remember, no other processing takes place while the interrupt is being serviced.

You can stop the timer interrupt with `STOP-TMRC2`.

```
\ 1 MILLISECOND INTERRUPT EXAMPLE

\ Count for 1 msec at 5 MHz timer clock
DECIMAL 5000 CONSTANT TMRC2_COUNT
HEX

\ Timer C2 registers
0D50 CONSTANT TMRC2_CMP1
0D53 CONSTANT TMRC2_LOAD
0D56 CONSTANT TMRC2_CTRL
0D57 CONSTANT TMRC2_SCR

\ GPIO interrupt control register
FFFB CONSTANT GPIO_IPR
2000 CONSTANT GPIO_IPL_2    \ bit which enables Channel 2
IPL

\ Interrupt vector & control.
\ Timer C channel 2 is vector 36, IRQ table address $48
0048 7D80 + CONSTANT TMRC2_VECTOR

\ Timer C channel 2 is controlled by Group Priority Register
GPR9, bits 2:0
\ Timer will use interrupt priority channel 2
0E69 CONSTANT TMRC2_GPR
0007 CONSTANT TMRC2_PLR_MASK
0003 CONSTANT TMRC2_PLR_PRIORITY  \ priority channel 2 in
bits 2:0

\ Initialize Timer C2
: START-TMRC2

    \ Set compare 1 register to desired # of cycles
    TMRC2_COUNT TMRC2_CMP1 !

    \ Set reload register to zero
    0 TMRC2_LOAD !

    \ Timer control register
    \  001 = normal count mode
    \     1 011 = IPbus clock / 8 = 5 MHz timer clock
    \          0 0 = secondary count source n/a
    \             0 = count repeatedly
    \              1 = count until compare, then reinit
    \               0 = count up
    \                 0 = no co-channel init
    \                  000 = OFLAG n/a
    \  0011 0110 0010 0000 = $3620
    3620 TMRC2_CTRL !

    \ Timer status & control register
    \ Clear TCF flag, set interrupt enable flag
    8000 TMRC2_SCR CLEAR-BITS
    4000 TMRC2_SCR SET-BITS

    \ Interrupt Controller
    \ set the interrupt channel = 3 for Timer D3
    TMRC2_PLR_MASK     TMRC2_GPR CLEAR-BITS
    TMRC2_PLR_PRIORITY TMRC2_GPR SET-BITS

    \ enable that interrupt channel in processor status
register
    GPIO_IPL_2 GPIO_IPR SET-BITS
;


\ Stop Timer C2
: STOP-TMRC2
    \ Timer control register
    \  000x xxxx xxxx xxxx = no count
    E000 TMRC2_CTRL CLEAR-BITS

    \ Timer status & control register
    \ Clear TCF flag, clear interrupt enable flag
    C000 TMRC2_SCR CLEAR-BITS
;

VARIABLE TICKS

\ High level word to handle the timer C2 interrupt
: TMRC2-IRPT
    \ clear the TCF flag to clear the interrupt
    8000 TMRC2_SCR CLEAR-BITS
    \ increment the ticks counter
    1 TICKS +!
;


HEX 0041 CONSTANT WP

CODE-SUB INT-SERVICE
DE0B P,           \  LEA  (SP)+
D00B P,           \  MOVE X0,X:(SP)+
D10B P,           \  MOVE Y0,X:(SP)+
D30B P,           \  MOVE Y1,X:(SP)+
D08B P,           \  MOVE A0,X:(SP)+
D60B P,           \  MOVE A1,X:(SP)+
D28B P,           \  MOVE A2,X:(SP)+
D18B P,           \  MOVE B0,X:(SP)+
D70B P,           \  MOVE B1,X:(SP)+
D38B P,           \  MOVE B2,X:(SP)+
D80B P,           \  MOVE R0,X:(SP)+
D90B P,           \  MOVE R1,X:(SP)+
DA0B P,           \  MOVE R2,X:(SP)+
DB0B P,           \  MOVE R3,X:(SP)+
DD0B P,           \  MOVE N,X:(SP)+
DE8B P,           \  MOVE LC,X:(SP)+
DF8B P,           \  MOVE LA,X:(SP)+
F854 P, OBJREF P, \  MOVE X:OBJREF,R0
FA54 P, WP P,     \  MOVE X:WP,R2
D80B P,           \  MOVE R0,X:(SP)+
DA1F P,           \  MOVE R2,X:(SP)   ; Note no increment on
last push!
87D0 P, ' TMRC2-IRPT CFA P,   \  MOVE #$XXXX,R0   ; CFA of
the word to execute
E9C8 P, ATO4 P,   \  JSR  ATO4        ; do that Forth word
FA1B P,           \  MOVE X:(SP)-,R2  ; restore the saved wp
F81B P,           \  MOVE X:(SP)-,R0  ; restore the saved
objref
FF9B P,           \  MOVE X:(SP)-,LA
DA54 P, WP P,     \  MOVE R2,X:WP
D854 P, OBJREF P, \  MOVE R0,X:OBJREF
FE9B P,           \  MOVE X:(SP)-,LC
FD1B P,           \  MOVE X:(SP)-,N
FB1B P,           \  MOVE X:(SP)-,R3
FA1B P,           \  MOVE X:(SP)-,R2
F91B P,           \  MOVE X:(SP)-,R1
F81B P,           \  MOVE X:(SP)-,R0
F39B P,           \  MOVE X:(SP)-,B2
F71B P,           \  MOVE X:(SP)-,B1
F19B P,           \  MOVE X:(SP)-,B0
F29B P,           \  MOVE X:(SP)-,A2
F61B P,           \  MOVE X:(SP)-,A1
F09B P,           \  MOVE X:(SP)-,A0
F31B P,           \  MOVE X:(SP)-,Y1
F11B P,           \  MOVE X:(SP)-,Y0
F01B P,           \  MOVE X:(SP)-,X0
EDD9 P,           \  RTI
END-CODE

\ Install the interrupt vector in Program Flash ROM
E984                  TMRC2_VECTOR PF!       \ JMP
instruction
' INT-SERVICE CFA 2+   TMRC2_VECTOR 1+ PF!    \ target
address
```

To install this interrupt you must have an IsoMax kernel version 0.5 or greater. This has a table of two-cell interrupt vectors starting at $7D80. The first cell (at $7D80+$48 for Timer C2) must be a machine-code jump instruction, $E984; the second cell is the address of the interrupt service routine. This address is obtained with the phrase `' INT-SERVICE CFA 2+` because the first two locations of a CODE-SUB or CODE-

INT are “overhead.” The interrupt vector is not installed with `EEWORD`; instead, it is programmed directly into Program Flash ROM with the `PF!` operator.

Observe also the use of `' TMRC2-IRPT CFA` to obtain the address “xxxx” of the high-level interrupt service routine.

This example is shown running out of Program RAM; that is, none of the words have been committed to Flash ROM with `EEWORD`. This is acceptable for testing, but for a real application you would want your interrupt handler to reside in ROM so that it survives a reset or a memory crash.

161. 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.

162. 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:

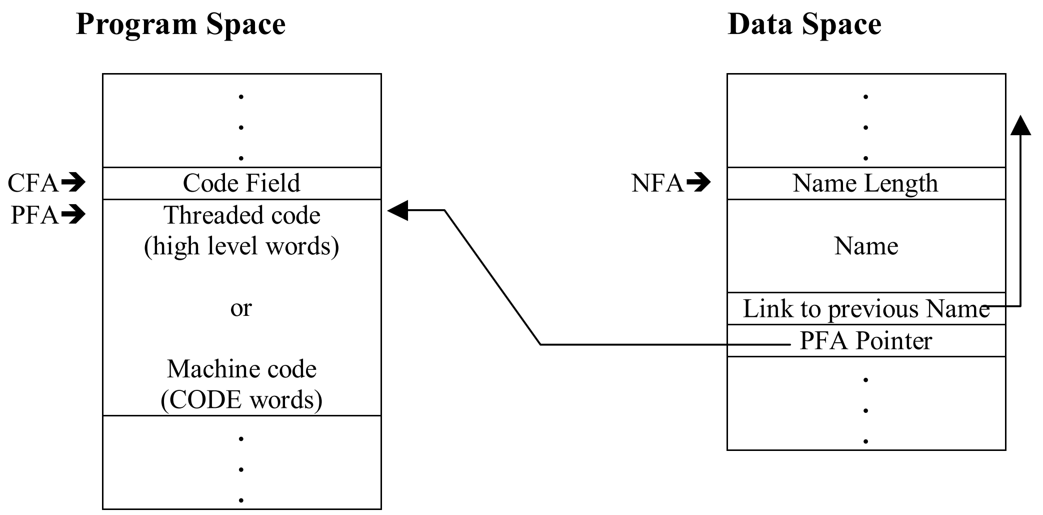
```
P@   P!   PC@   PC!   PHERE   PALLOT   P,   PC,
```

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.

163. WORD STRUCTURE

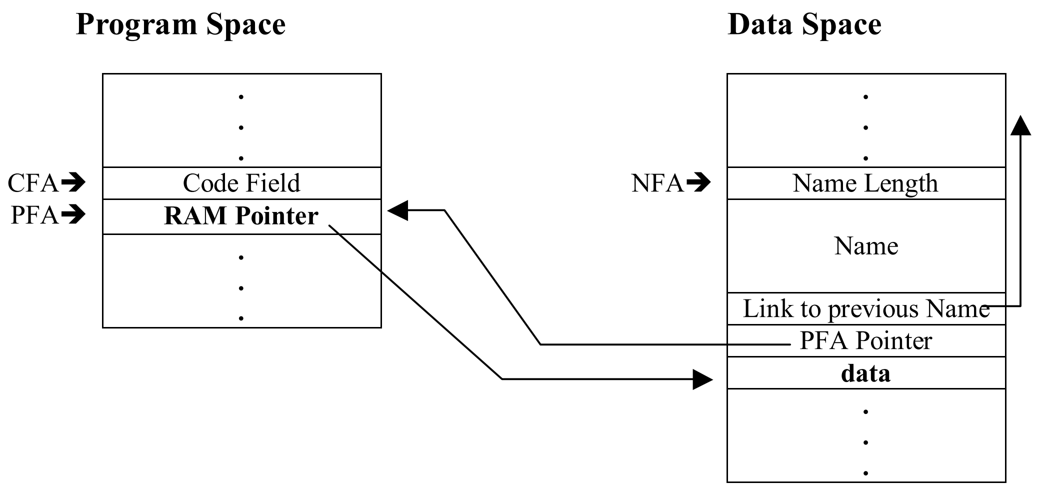
The executable "body" of a IsoMax™ word is kept in Program space. This includes the Code Field of the word, and the threaded definition of high-level words or the machine code definition of CODE words.

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



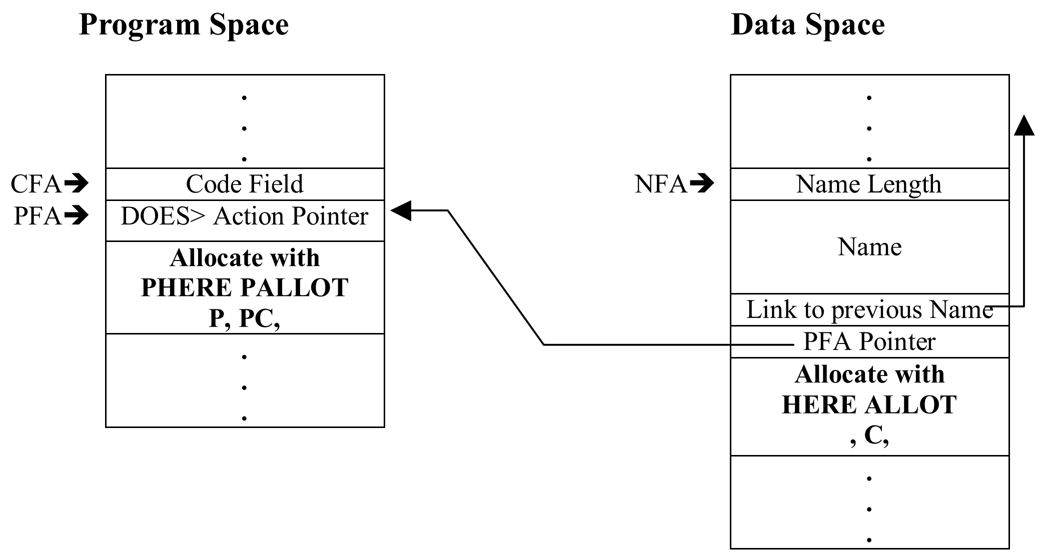
164. VARIABLES

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.



165. <BUILDS DOES>

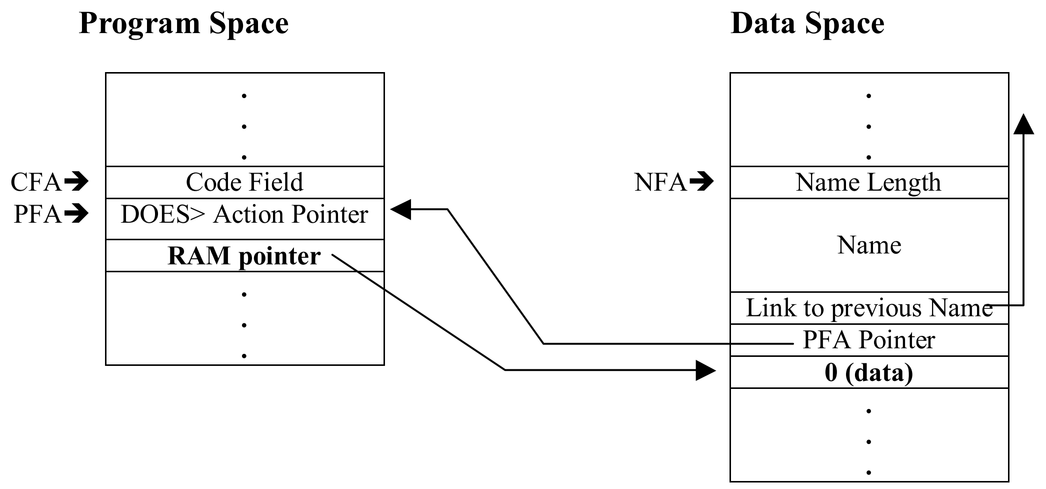
"Defining words" created with <BUILDS and DOES> may have a variety of purposes. Sometimes they are used to build Data objects in RAM, and sometimes they are used to build objects in ROM (i.e., in Program space). In the <BUILDS code you can allocate either space by using the appropriate memory operators.



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.
;
```

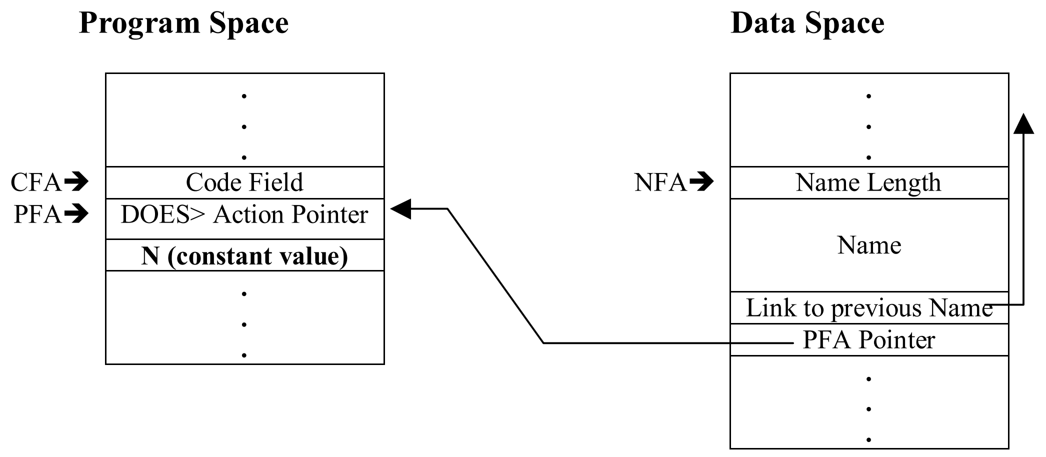
This constructs the following:



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:



166. Object Oriented Internals

For this illustration we will use the **BYTEIO** class from the file Gpioobj.4th (appended below).

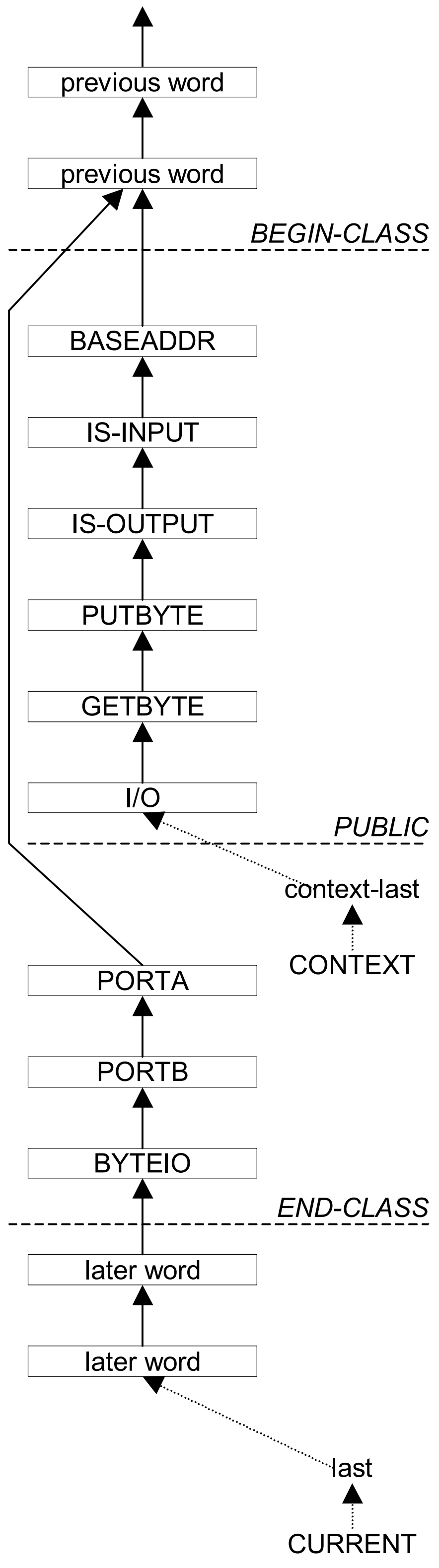
167. Dictionary Hiding

BEGIN-CLASS marks the start of definitions that will be "hidden." Once they are hidden, they will only be visible to members of this class. **BEGIN-CLASS** just marks a dictionary position; it doesn't compile anything.

PUBLIC marks the end of the hidden definitions. It does two things. First, it puts a pointer to the last-defined word (i.e., the last hidden word) in the **context-last** variable. This means these words will still be found when the **CONTEXT** list is searched. Second, it relinks the main dictionary list around the hidden words, by resetting the **last** variable.

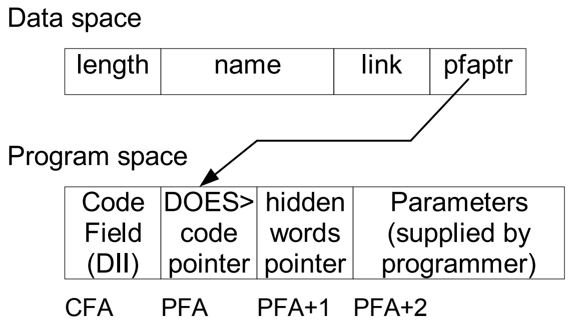
At this point, the hidden words are still searchable, and can still be used to write Forth definitions. New definitions will be "public" and will be part of the main dictionary list, not the hidden list.

END-CLASS hides the private definitions, by clearing **CONTEXT**. It also creates a class-name word (in this example, **BYTEIO**) which will make the private word list visible again, by putting its dictionary link back into the **context-last** variable.



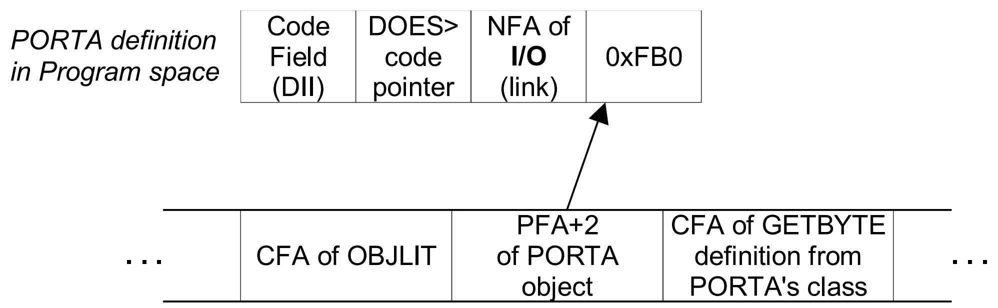
168. Object Action

A word created with **OBJECT** has both a compile-time action and a run-time action. At compile-time (or when interpreted), it makes its hidden word list visible, by putting the dictionary link into the **context-last** variable. Thus, after an object is named, its private "methods" can be compiled or interpreted.



At run-time, an object puts the address of its parameters (PFA+2) into the **OBJREF** variable. This is essentially the same as **DOES>**, except that the address is stored into a variable instead of being left on the stack. The "methods" which follow the object all expect to find this address in **OBJREF**. (The word **SELF** returns this address.)

Note: when an object is used in a Forth definition, what actually gets compiled is a *literal* (in-line constant) with the address PFA+2. Thus the phrase **PORTA GETBYTE** is compiled as



The special word **OBJLIT** takes the in-line value which follows, and stores it in the **OBJREF** variable. This is exactly the same as the Forth primitive **LIT**, except that the value is stored in a variable instead of being left on the stack.

In this example, the **PORTA** definition has one user-supplied parameter: the value 0xFB0, which is the I/O address of the desired port. The object is created, and this extra parameter is appended, by the word **I/O** (see below).

```
\ -----------------------------------------------------------
\ GPIO PARALLEL PORTS - BYTE I/O
\ -----------------------------------------------------------
BEGIN-CLASS BYTEIO

\ BYTEIO methods expect SELF to point to:  baseaddr  in ROM
: BASEADDR ( -- a )   SELF P@ ;

: IS-INPUT    ( makes pin an input
    0FF  BASEADDR 3 + CLEAR-BITS   ( PER=0, GPIO
    0FF  BASEADDR 2+  CLEAR-BITS   ( data dir=in
;

: IS-OUTPUT   ( makes pin an output
    0FF  BASEADDR 3 + CLEAR-BITS   ( PER=0, GPIO
    0FF  BASEADDR 2+  SET-BITS     ( data dir=out
;

: PUTBYTE ( c -- )   IS-OUTPUT  BASEADDR 1+ C! ;
: GETBYTE ( -- c )   IS-INPUT   BASEADDR 1+ C@ ;

\ define an I/O port
: I/O ( baseaddr -- )   OBJECT  P, ;

PUBLIC

FB0 I/O PORTA
FC0 I/O PORTB

END-CLASS BYTEIO
```

169. CPU 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

170. 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

171. 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.

172. 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)

173. 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

174. CAN BUS Levels Translation

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

175. 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

176. 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

177. 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

178. 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.

179. REFERENCE

180. IsoPod™ website:

http://www.isopod.net

181. MaxFORTH™ Glossary Reference Page

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

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

182. Motorola DSP56F805 Users Manual

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

183. Motorola DSP56F800 Processor Reference Manual

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

184. Appendix: DSP56F805 Instruction Encoding

```
                  DSP56800 OPCODE ENCODING

(1)    00Wk  kHHH  Fjjj  xmRR  (14)     P1DALU  jjj,F X:<ea_m>,HHH

(2)    010y  y0yy  y*pp  pppp  (11-*)  ADD/SUB/CMP/INC/DEC X:<aa>[,fff]
(3)    010y  y0yy  y+aa  aaaa  (11-*)  ADD/SUB/CMP/INC/DEC X:(SP-xx)[,fff]
(4)    010y  y1yy  y00B  BBBB  (10)     ADD/SUB/CMP #<0-31>,fff
(5a)   010y  y1yy  y10-  ----  (5-2)    ADD/SUB/CMP #xxxx,fff
(5b)   010y  y1yy  yw11  -1--  (6-2)    ADD/SUB/CMP/INC/DEC X:xxxx[,fff]

(7)    011u  u0v1  Fvjj  xm-v  (10)     P2DALU jj,F   X:<ea_m>,reg X:<ea_v>,X0
(8a)   011L  L1L-  FQQQ  10FF  (9)      DALU3OP  QQQ,FFF
(8b)   011I  I1II  FQQQ  11FF  (10)     DALU3OP2 QQQ,FFF
(8c)   011K  K1K-  F000  0h00  (4)      DALU2OPF  ~F,F   (KKK = KK0)  (h=1: Tcc)
(8d)   011K  K1K-  F000  0h00  (4)      DALU2OPY   Y,F   (KKK = KK1)  (h=1 used)
(8e)   011K  K1K-  F000  0hF1  (5)      DALU2OPB1  B1,FF          (h=1: Tcc)
(8f)   011K  K1K-  F010  0hF1  (5)      DALU2OPA1  A1,FF          (h=1: Tcc)
(8g)   011K  K1K-  F0qq  0h00  (6)      DALU1OPF   F  (qq != 00)   (h=1 used)
(8h)   011K  K1K-  F0q1  0hF1  (6)      DALU1OPFF  FF         (h=1: LSL,LSR)
(8i)   011K  K1K-  F1JJ  0hFF  (8)      DALU2OPJJ  JJ,FFF     (h=1: DIV,Tcc)
(8j)   0110  11CC  FJJJ  01CZ  (8)      Tcc       JJJ,F   [R0->R1]   (h=1: Tcc)

(9)    10W1  HHHH  0Ppp  pppp  (12)     MOVE X:<Ppp>,REG
(10a)  10W1  HHHH  1*AA  AAAA  (11)     MOVE X:(R2+xx),REG
(10b)  10W1  HHHH  1+aa  aaaa  (11)     MOVE X:(SP-xx),REG
(11)   11W1  DDDD  D0-M  RMRR  (12)     MOVE X:<ea_MM>,DDDDD
(12)   11W1  DDDD  D1-0  R1RR  (10)     MOVE X:(Rn+N),DDDDD
(13)   11W1  DDDD  D1-0  R0RR  (10-2)  MOVE X:(Rn+xxxx),DDDDD
(14)   11W1  DDDD  D1-1  -1--  (7-2)    MOVE X:<abs_adr>,DDDDD

(15)   1000  DDDD  D00d  dddd  (10)     MOVE ddddd,DDDDD
(16)   1000  1110  *011  00RR  (2)      TSTW (Rn)-
(17)   1000  UUU+  110d  dddd  (8-2)    BITFIELD DDDDD;     MOVE #xxxx,DDDDD
(18)   1000  UUU0  111+  -+--  (3-3)    BITFIELD X:xxxx;    MOVE #xxxx,X:xxxx
(19a)  1010  UUU0  1+aa  aaaa  (9-2)    BITFIELD X:(SP-xx); MOVE #xxxx,X:(SP-xx)
(19b)  1010  UUU0  1*AA  AAAA  (9-2)    BITFIELD X:(R2+xx); MOVE #xxxx,X:(R2+xx)
(20)   1010  UUU1  1Ppp  pppp  (10-2)  BITFIELD X:<Ppp>;   MOVE #xxxx,X:<Ppp>
(21)   1010  CCCC  0Aaa  aaaa  (11)     Bcc <aa>,  BRA

(22)   1100  HHHH  *BBB  BBBB  (11)     MOVE #xx,HHHH
(23)   1100  11E0  1*BB  BBBB  (7-*)    DO/REP #xx
(24)   1100  11E0  11-d  dddd  (6-*)    DO/REP ddddd
(25a)  1110  CCCC  10A-  -1AA  (7-2)    Jcc, JMP  xxxxx
(25b)  1110  1001  11A0  10AA  (*-2)    JSR  xxxxx
(26)   1110  1101  11-1  10-0  (0)      RTS
(27)   1110  1101  11-1  10-1  (0)      RTI
(29)   1110  HHHH  *0W*  *mRR  (8)      MOVE P:<ea_m>,HHHH

(30)   1110  ----  -1--  0000  (0)      NOP
(31)   1110  ----  -1--  0001  (0)      DEBUG
(--)   1110  ----  -1--  0010  (0)      ($E042 -reserved for "ADD <reg>,<mem>")
(32)   1110  ----  -1--  01tt  (2)      STOP, WAIT, SWI,  ILLEGAL

(--)   1100  ----  111-  ----  (9)      <Available Hole>
(--)   1110  ----  111-  ----  (9)      <Available Hole>
(--)   1110  ----  01--  ----  (10)     <Available Hole>

Understanding entries in the above encoding:
--------------------------------------------
A typical entry in the encoding files looks like this:

(8b)   011I  I1II  FQQQ  11FF  (10)     DALU3OP2 QQQ,FFF
```

```
 ^     \                    /   ^         ^
 |      ---------v----------    |          |
 |               |              |          |
 |               |              |          +---- (see #1 below)
 |              |             +------------- (see #2 below)
 |             +----------------------------- (see #3 below)
 +---------------------------------------------- (see #4 below)
```

#1: This field gives the name of the instruction or of a class of instructions which are encoded with the bit pattern specified in #3.

An example of where this field contains an instruction is for the "TSTW (Rn)-" instruction. In this case, only the operands of the instruction are encoded with the bits in #3 below.

An example of where this field contains a class of instructions is given in the example above "DALU3OP2 QQQ,FFF". In this case, the entry DALU3OP2 represents a class of instructions, and the instruction selected within this class is selected by the IIII field within the encoding specified in #2.

Instruction classes such as "DALU3OP2" can be seen by searching in this file for the following field - "DALU3OP2:", where the field is located in the very first character of the line.

#2: The number here indicates how many bits are required to encode this instruction. For the example shown above, 10 bits are required to hold the following bits - IIIIFQQQFF. The information in this particular field is useful to the design group.

If the number in this field is followed by a "-2" or "-3", the "-2" is used to indicate a two word instruction, and the "-3" is used to indicate a three word instruction.

For the case of the "ADD/SUB/CMP/INC/DEC X:<aa>[,fff]" instruction which uses "(11-*)", this indicates that this class of instructions can vary in number of instruction words. For this particular example, this can be seen more clearly in the section entitled "Unusual Instruction Encodings" located within this document.

#3: This portion represents the 16 opcode bits of the instruction. For single word instructions, it contains the entire one word 16-bit opcode.   For multiword instructions, it contains the first word for the instruction.

The example above contains the following fields within the instruction:
  IIII, FFF, QQQ
Note that although there are four I bits to form the "IIII" field, these bits are not necessarily all next to each other. This is also the case for the three bits comprising the "FFF" field.

#4: The number here gives a unique number to this particular instruction or class of instructions. This is used simply for identification purposes.

Notes for Above Encoding:
----

1. Where a "*" is present in a bit in the encoding, this means the PLAs often use this bit to line up in a field, but that the assembler should always see this as a "0". Where a "+" is present, it is similar, but assembles as a "1". A "-" is ignored by the PLAs and assembled as a "0".

2. It is important to note that several instructions are not found on the first page of the encoding, which summarizes the entire instruction set. These instructions are instead found in the section entitled "Unusual Instruction Encodings" located within this document. Instructions in this section include:
   - ADD fff,X:<aa>:

- ADD fff,X:(SP-xx):
- ADD fff,X:xxxx:
- LEA
- TSTW
- POP
- CLR   (although CLR is also encoded in the Data ALU section)
- ENDDO

See this section to see how these instructions are encoded.

3. The use of the bit pattern labelled
   "($E042 -reserved for "ADD <reg>,<mem>")"
   is explained in more detail in the "Unusual Instruction Encodings" section. It is not an instruction in itself, but rather enables an encoding trick discussed for the ADD instruction in that section.

Understanding the 2 and 1 Operand Data ALU Encodings

The Data ALU operations were encoded in a manner which is not straightforward. The three operand instructions were relatively straightforward, but the encoding of the two and one operand instructions was more difficult.

More information is presented at the field definitions for "KKK" and "JJJ". This is the best place to clearly understand the Data ALU encodings.

(Also see the encoding information located at the "KKK" field.)

Data ALU Source and Destination Register Field Definitions:

F:

F	Destination Accumulator
0	A
1	B

~F:

"~F" is a unique notation used in some cases to signify the source register in a DALU operation. It's exact definition is as follows:
If "F" is the "A" accumulator, Then "~F" is the "B" accumulator.
If "F" is the "B" accumulator, Then "~F" is the "A" accumulator.

FF:

FF	Destination Register
00	X0 (NOTE: not all DALU instrs can have this as a destination)
10	(reserved)
01	Y0 (NOTE: not all DALU instrs can have this as a destination)
11	Y1 (NOTE: not all DALU instrs can have this as a destination)

FFF:

FFF	Destination Register
000	A
100	B
001	X0 (NOTE: not all DALU instrs can have this as a destination)
101	(reserved)
011	Y0 (NOTE: not all DALU instrs can have this as a destination)
111	Y1 (NOTE: not all DALU instrs can have this as a destination)

NOTE: The MPY, MAC, MPYR, and MACR instructions allow x0, y0, or y1 as a destination. FFF=FF1 IS allowed for the case of a negated product: -y0,x0,FFF for example is allowed. Also, MPYsu, MACsu, IMPY16, LSRR, ASRR, and ASLL allow FFF as a destination, but the ASRAC & LSRAC instructions only allow F, and LSLL only allows DD as destinations.

Although the LSLL only allows 16-bit destinations, there is the ASLL instruction which performs exactly the same operation and allows an accumulator as well as a destination.

fff:

fff	Destination Register	
000	A	(ADD/SUB/CMP only)
001	B	(ADD/SUB/CMP only)
100	X0	(ADD/SUB/CMP only)
101	(reserved for X1)	
110	Y0	(ADD/SUB/CMP only)
111	Y1	(ADD/SUB/CMP only)

QQQ: (6-4)

This field specifies two input registers for instructions in the DALU3OP, DALU3OP2, and P1DALU instruction classes. There are some instructions where the ordering of the two source operands is important and some where the ordering is unimportant.

Three different cases are presented below for instructions using the QQQ field. Some examples are also included for clarification. Note that the bottom 4 entries are designed to overlay the "QQ" field.

1. "QQQ" definition for: ASRR, ASLL, LSRR, LSLL, ASRAC, & LSRAC instrs

QQQ	Shifter inputs (must be in this order)
000	(reserved for X1,Y1)
001	B1,Y1
010	Y0,Y0
011	A1,Y0
100	Y0,X0
101	Y1,X0
110	(reserved for X1,Y0)
111	Y1,Y0

For Multi-bit shift instructions:
- 1st reg specified is value to be shifted
- 2nd reg specified is shift count (uses 4 LSBs)

Examples of valid Multi-bit shift instructions:
```
  asll b1,y1,a ; b1 is value to be shifted, y1 is shift amount
  asrr y1,x0,b ; y1 is value to be shifted, x0 is shift amount
```

Examples of INVALID Multi-bit shift instructions:
```
  asll y1,b1,a ; Not allowed - b1 must be first for QQQ=001
  asrr x0,y1,b ; Not allowed - y1 must be first for QQQ=101
```

2. "QQQ" definition for: MPYsu and MACsu instrs

QQQ	Multiplier inputs (must be in this order)
000	(reserved for Y1,X1)
001	Y1,B1
010	Y0,Y0
011	Y0,A1
100	X0,Y0
101	X0,Y1
110	(reserved for Y0,X1)
111	Y0,Y1

For MPYsu or MACsu instructions:
- 1st reg specified in QQQ above is   "signed" value
- 2nd reg specified in QQQ above is "unsigned" value

Examples of valid MPYsu and MACsu instructions:
```
  mpysu y1,b1,a   ; y1 is signed, b1 unsigned, QQQ = 001
  macsu x0,y1,b   ; x0 is signed, y1 unsigned, QQQ = 101
```

Examples of INVALID MPYsu and MACsu instructions:

```
  mpysu b1,y1,a   ; Not allowed - y1 must be signed for QQQ=001
  macsu y1,x0,b   ; Not allowed - x0 must be signed for QQQ=101
```

The Multi-bit shift instructions include:
ASRR, ASLL, LSRR, LSLL, ASRAC, and LSRAC

3. "QQQ" definition for: All other instructions using "QQQ"

QQQ	Multiplier inputs	Also Accepted by Assembler
000	(reserved for Y1,X1)	(reserved for X1,Y1)
001	Y1,B1	B1,Y1
010	Y0,Y0	Y0,Y0
011	Y0,A1	A1,Y0
100	X0,Y0	Y0,X0
101	X0,Y1	Y1,X0
110	(reserved for Y0,X1)	(reserved for X1,Y0)
111	Y0,Y1	Y1,Y0

For all other of these instructions:
- operands can be specified in either order

Examples of valid MPY and MAC instructions:

```
  mpy   y1,b1,a   ; Operands are: y1 and b1   (ordering unimpt)
  mpy   b1,y1,a   ; Operands are: y1 and b1   (ordering unimpt)
  mac   x0,y1,b   ; Operands are: y1 and x0   (ordering unimpt)
  mac   y1,x0,b   ; Operands are: y1 and x0   (ordering unimpt)
```

NOTE: If the source operand ordering is incorrect, then the assembler must flag this as an error.

Data-Alu Opcode Field Definitions:

q: used to specify "non-multiply" one operand DALU/P1DALU instructions. See the "KKK" field definition below.

qq: used to specify "non-multiply" one operand DALU/P1DALU instructions. See the "KKK" field definition below.

DALU3OP:

LLL:

LLL	Multiplication Operation
000	MPY + (neither operand inverted)
001	MPY - (one operand inverted)
010	MAC + (neither operand inverted)
011	MAC - (one operand inverted)
100	MPYR + (neither operand inverted)
101	MPYR - (one operand inverted)
110	MACR + (neither operand inverted)
111	MACR - (one operand inverted)

h: (2)
The "h" bit, when set to a "1" is used to encode the following non-multiply DALU instructions:
- ADC, SBC
- NORM R0
- LSL, LSR
- DIV

For exact details on this, see the "KKK" field definition below.

DALU2OPF:
DALU2OPY:
DALU2OPB1:

DALU2OPA1:
DALU1OPF:
DALU1OPFF:
DALU2OPJJ:

KKK: ()

The KKK fields cannot be uniquely decoded without looking at the values in some other bits of the opcode. In the below charts, the KKK field holds many different encodings depending on the values in bits 6-4, what was previously called the JJJ field, and bit 2, which was previously labelled as "h". The JJJ and h fields have now been removed and this chart now contains the information previously held by these bits.

Four different charts are presented below, where the four charts correspond to different values "00, 01, 10, and 11" in bits 2 and 0 of the opcode.

Note that the KKK entries are numbered in an ascending order from 0 to 7. This also differs from the numbering in the original encoding file (encode8) so the entries in the chart will now appear to be in a different order.

Notation for the below charts:

```
  <<NA>>    - Indicates field is not available for any instruction
  <<Tc>>    - Indicates space is not available because it is occupied
              by the Tcc instruction.
   ~F      - Indicates source is the accumulator not used as the dest
   ---      - Indicates field is unused
```

Chart 1 - Basic Data ALU, Destination is "F"

This chart is used to encode MOST non-multiply Data ALU instructions where the result of the operation is stored in one of the accumulators, A or B, i.e. is of the form "NONMPY_DALUOP <src>,F".

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "F" bit.

bbb b b					KKK				
iii i i					---				
ttt t t									
... . .									
654 2 0									
KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KK0 000 0 0	~F	ADD	<<NA>>	TFR	<<NA>>	SUB	<<NA>>	CMP	<<NA>>
KK1 000 0 0	Y	<<NA>>	ADD	<<NA>>	--	<<NA>>	SUB	<<NA>>	--
KKK 001 0 0	F	DECW	--	NEG	NOT	RND	--	TST	--
KKK 010 0 0	F	--	--	ABS	--	--	--	--	--
KKK 011 0 0	F	INCW	--	CLR	--	ASL	ROL	ASR	ROR
KKK 100 0 0	X0	ADD	OR	TFR	--	SUB	AND	CMP	EOR
KKK 101 0 0	Y0	ADD	OR	TFR	--	SUB	AND	CMP	EOR
KKK 110 0 0	--	--	--	--	--	--	--	--	--
KKK 111 0 0	Y1	ADD	OR	TFR	--	SUB	AND	CMP	EOR

Note that there are nine rows above. This is because the entry for "JJJ" = 000 is broken into two different rows - one where the LSB of "KKK" is "0" (source is "~F") and one row where the LSB is "1" (source is "Y") .

Chart 2 - Basic Data ALU, Destination is "DD"

This chart is used to encode MOST non-multiply Data ALU instructions where the result of the operation is stored in one of the data regs, X0, Y0 or Y1, i.e. is of the form "NONMPY_DALUOP <src>,DD".

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "FF" bits.

bbb b b		KKK							
iii i i		---							
ttt t t		|							
... . .		|							
654 2 0		|							
KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KKK 000 0 1	B1	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 001 0 1	F	DECW	--	--	NOT	--	--	--	--
KKK 010 0 1	A1	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 011 0 1	F	INCW	--	--	--	*	ROL	ASR	ROR
KKK 100 0 1	X0	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 101 0 1	Y0	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 110 0 1	--	--	--	--	--	--	--	--	--
KKK 111 0 1	Y1	ADD	OR	--	--	SUB	AND	CMP	EOR

* For 16-bit destinations, "asl" is identical to "lsl". Thus, if a user has "asl x0" in his program, it should instead assemble into "lsl x0". Always disassembles as "lsl x0".

Chart 3 - Supplemental Data ALU, Destination is "F"

This chart is used to encode A FEW non-multiply Data ALU instructions where the result of the operation is stored in one of the accumulators, A or B, i.e. is of the form "NONMPY_DALUOP <src>,F".

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "F" bit.

bbb b b		KKK							
iii i i		---							
ttt t t		|							
... . .		|							
654 2 0		|							
KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KK0 000 1 0	~F	--	<<NA>>	<<Tc>>	<<NA>>	--	<<NA>>	--	<<NA>>
KK1 000 1 0	Y	<<NA>>	ADC	<<NA>>	<<Tc>>	<<NA>>	SBC	<<NA>>	--

KKK 001 1 0	F	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 010 1 0	F	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 011 1 0	F	--	--	<<Tc>>	<<Tc>>	--	LSL	NORM	LSR
KKK 100 1 0	X0	DIV	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 101 1 0	Y0	DIV	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 110 1 0	--	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 111 1 0	Y1	DIV	--	<<Tc>>	<<Tc>>	--	--	--	--

Note that there are nine rows above. This is because the entry for "JJJ" = 000 is broken into two different rows - one where the LSB of "KKK" is "0" (source is "~F") and one row where the LSB is "1" (source is "Y") .

Tcc instructions that occupy space on this chart are Tcc instructions where the "Z" bit is a "0". This corresponds to Tcc instructions of the form "tcc <reg>,F", i.e., without an AGU register transfer.

Chart 4 - Supplemental Data ALU, Destination is "DD"

This chart is used to encode A FEW non-multiply Data ALU instructions where the result of the operation is stored in one of the data regs, X0, Y0 or Y1, i.e. is of the form "NONMPY_DALUOP <src>,DD".

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "FF" bits.

bbb b b iii i i ttt t t 654 2 0		KKK ---							
KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KK0 000 1 1	B1	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 001 1 1	DD	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 010 1 1	A1	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 011 1 1	DD	--	--	<<Tc>>	<<Tc>>	--	LSL	--	LSR
KKK 100 1 1	X0	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 101 1 1	Y0	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 110 1 1	--	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 111 1 1	Y1	--	--	<<Tc>>	<<Tc>>	--	--	--	--

Tcc instructions that occupy space on this chart are Tcc instructions where the "Z" bit is a "1". This corresponds to Tcc instructions of the form "tcc <reg>,F r0,r1", i.e., with an AGU register transfer.

yyyyy:
The "yyyyy" field is used to determine the operand encoding and destination operand definitions for data ALU instructions where one source operand

```
   is not a Data ALU register.  It is described as "010" type instructions
   because all instructions in this class begin with "010" in bits 15-13.

   For instructions of this type, the destination is always specified with
   the "fff" field.

       yyyyy  Operation
       -----  ------------
       00fff  ADD <src>,fff
       10fff  SUB <src>,fff
       11fff  CMP <src>,fff
       01100  DEC <dst>     NOTE: src and dst is a memory location, not a reg
       01101  INC <dst>     NOTE: src and dst is a memory location, not a reg
       0111x  <Available>

       DALU3OP2 - Shifting and Multiplication Encoding Information
       --------------------------------------------------------

DALU3OP2:
---------
IIII: ()
       Specifies Integer Multiplication, Signed*Uns, and Shifting Instructions

              IIII    Operation
              ----   ---------
              1000    MPYsu
              1100    MACsu
              0010    IMPY16
              1001    LSRR        (multibit logical right shift)
              1101    LSRAC       (used for shifting 32-bit values)
              0001    ASRR        (multibit arithm  right shift)
              0101    ASRAC       (multibit arithm  right shift w/ acc)
              0011    ASLL or LSLL(multibit arithm  left  shift)
              ----
              ^^^^
              ||||
              |||+--- Indicates no shifting or shifting
              ||+---- Shift shift dirn and whether LSP goes to DXB1
              |+----- Selects mpy vs mac operation
              +------ Selects signed*signed vs signed*unsigned

     Note: no inversion of multiplier result or rounding is allowed.

     NOTE: All of the above allow FFF as a destination EXCEPT
          LSRAC and ASRAC which only allow F as a destination,
          and LSLL which only allows X0, Y0, and Y1 as destinations.

          Although the LSLL only allows 16-bit destinations, there is
          the ASLL instruction which performs exactly the same operation
          and allows an accumulator as well as a destination.

Single Parallel Move Encodings:
===============================

P1DALU:
-------
x:
kk:
jjj:
      P1DALU operation and source register encodings (xkkjjj)
       x kk jjj
       - -- ---
       0 KK JJJ   - KK  specifies the arithm operation for non-multiply instrs
                 - JJJ specifies one source operand   for non-multiply instrs
                      (kk becomes KK when x=0)
                      (jjj becomes JJJ when x=0)
       1 LL QQQ   - LL  specifies the arithm operation for  multiply instrs
                 - QQQ specifies one source operand   for    multiply instrs
```

(kk becomes LL when x=1)
(jjj becomes QQQ when x=1)

JJJ:
Specifies the source registers for the "non-multiply" P1DALU class of instructions as well as the Tcc instruction.

JJJ	Source register	
000	~F	
001	F	(not used by the Tcc instruction)
01x	F	(not used by the Tcc instruction)
01x	F	(not used by the Tcc instruction)
100	X0	
101	Y0	
110	(reserved for X1)	
111	Y1	

KK: ()

Chart 5 - Single Parallel Move Data ALU, Destination is "F"

This chart is used to encode all of the non-multiply arithmetic operations with a SINGLE PARALLEL MOVE, where the result of the operation is stored in one of the accumulators, A or B. In this case, the instruction is of the following form

```
"NONMPY_DALUOP  <src>,F    <single_pll_mov>"
```

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "F" bit.

bbb iii ttt ... 654		KK --			
KK JJJ	SRC	00	01	10	11
KK 000	~F	ADD	TFR	SUB	CMP
KK 001	F	DECW	NEG	RND	TST
KK 010	F	--	ABS	--	--
KK 011	F	INCW	CLR	ASL	ASR
KK 100	X0	ADD	TFR	SUB	CMP
KK 101	Y0	ADD	TFR	SUB	CMP
KK 110	--	--	--	--	--
KK 111	Y1	ADD	TFR	SUB	CMP

Note that this chart is simply extraced from the above chart where bit_2 == 0 and bit_0 == 0. In this case, only the even values within the "KKK" field are retained.

Dual Parallel Read Encodings:

P2DALU:

x: ()

```
uu: ()
jj: ()
      P2DALU operation and source register encodings (xuujj)
       x uu jj
       - -- --
       0 UU GG   - UU specifies the arithm operation for non-multiply instrs
               - GG specifies one source operand   for non-multiply instrs
                    (uu becomes UU when x=0)
                    (jj becomes GG when x=0)
       1 LL QQ   - LL specifies the arithm operation for     multiply instrs
               - QQ specifies one source operand   for     multiply instrs
                    (uu becomes LL when x=1)
                    (jj becomes QQ when x=1)

GG: ()
UU: ()
     Specifies "non-multiply" P2DALU instructions and operands.
       x UU GG     Non-Multiply Operation       DALU Source Register
       - -- --     ----------------------       --------------------
       0 00 JJ      ADD                   JJ
       0 10 JJ      SUB                   JJ
       -------
       0 01 --      MOVE                  <none>
       -------
       0 11 --      (reserved)            <none>

JJ: ()
     Specifies the source registers for the "non-multiply" P2DALU instructions.
       JJ     source register
       --     ---------------
       00     X0
       01     Y0
       10     (reserved for X1)
       11     Y1

LL: ()
       LL     Multiplication Operation
       --     ------------------------
       00      MPY  +   (neither operand inverted)
       01      MAC  +   (neither operand inverted)
       10      MPYR +   (neither operand inverted)
       11      MACR +   (neither operand inverted)

QQ: ()
     Input registers for the "multiply" P2DALU instructions.
       QQ   Multiplier inputs
       --   -----------------
       00   Y0,X0
       01   Y1,X0
       10   (reserved for X1,Y0)
       11   Y1,Y0

vvv: (9,6,0)
    Specifies the destination registers for the dual X memory
    parallel read instruction WITH arithmetic operation.

       vvv    1st read     2nd access
       ---    --------     ----------
       000    X:(R0),Y0     X:(R3)+,X0      -
       010    X:(R0),Y0     X:(R3)-,X0      -
       100    X:(R0),Y1     X:(R3)+,X0      -
       110    X:(R0),Y1     X:(R3)-,X0      -

       001    X:(R1),Y0     X:(R3)+,X0      -
       011    X:(R1),Y0     X:(R3)-,X0      -
       101    X:(R1),Y1     X:(R3)+,X0      -
       111    X:(R1),Y1     X:(R3)-,X0      -
       ---
```

```
^^^
|||
||+---  (effectively an "r" bit for 1st read - R0 vs R1)
|+----  (effectively an "m" bit for 2nd read - (R3)+ vs (R3)-)
+-----  (effectively a  "V" bit for 1st read - Y0 vs Y1)
```

NOTE: Above table does not show any addressing mode information for the 1st read. See the "m" field for this information. The above table does contain addressing mode info for the second access as seen above.

Move Register Field Definitions:
================================

HHH: destination registers for the "P1DALU X:<ea_m>,HHH" instruction.

HHH	register
000	X0
001	Y0
010	(reserved for X1)
011	Y1
100	A
101	B
110	A1
111	B1

RRR: ()

RRR	register
000	R0
001	R1
010	R2
011	R3
111	SP

HHHH: destination registers for the "#xx,HHHH" instruction.

HHHH	register
0HHH	X0, Y0, (reserved for X1), Y1, A, B, A1, B1
10RR	R0, R1, R2, R3
11NN	ND (dst only), N, NOREG (src and dst), (reserved)

DDDDD: - specifies destination registers for "ddddd,DDDDD"
- specifies source/destination registers for other DDDDD moves
- NOTE that ordering is different than "ddddd"

DDDD	D	register
0HHH	0	X0, Y0, (reserved for X1), Y1, A, B, A1, B1
10RR	0	R0, R1, R2, R3
11xx	0	ND (dst only), N, NOREG, (reserved)
00xx	1	A0, B0, A2, B2
01xx	1	M01, (res), (res), SP
1xxx	1	OMR, PINC/PAMAS, (res), HWS, (res, used as LC), SR, LC, LA

ddddd: - specifies source registers for the move ddddd,DDDDD instruction.
- specifies source registers for the DO/REP ddddd instruction.
- specifies source/destination registers for bitfield instructions
- NOTE that ordering is different than "DDDDD"

ddddd	register
00HHH	X0, Y0, (reserved for X1), Y1, A, B, A1, B1
100RR	R0, R1, R2, R3
101xx	(res-ND), N, (res-NOREG), (res)
010xx	A0, B0, A2, B2
011xx	M01, (res), (res), SP
11xxx	OMR, PINC/PAMAS, (res), HWS, (res, used as LC), SR, LC, LA

```
Special registers which need to be detected:
         1110 0   NOREG   - Prevents external bus cycle, or perhaps any
                          memory cycle from occurring.  Required because
                          the chip may not own the bus.  Forces access
                          internal, or perhaps even disables prxrd/prxwr.
                          Occurs on read from reg only.  Note there is
                          no register actually present.  It applies to
                          reads from the register because this is true
                          during an LEA where no memory cycle is desired,
                          but this is not true for a TSTW instruction,
                          which must actually perform a memory cycle
                          and move the data onto the cgdb.
         1100 0   ND      - Accesses "N" register but also asserts pmnop.
                          Occurs on write to reg only.
         1100 0   ND      - Prevents interrupts, force adr onto eab,
                          regardless of whether it's on-chip or not.
                          Note there is no actual register.  Asserts
                          a new ctrl signal, pmdram.  Occurs on reads
                          from reg only.  Used to be the DRAM register.
                          Must disable xmem writes, similar to reads
                          from NOREG.  Force the access internal.
         1011 1   HWS     - Any reads of this register must "pop" the
                          HWS and HWSP.  Any writes to this register
                          must "push" the HWS and HWSP.

RR:    RR     register
       --     --------
       00     R0
       01     R1
       10     R2
       11     R3

AGU (Address Generation Unit) Instruction Field Definitions:
============================================================

MM: specifies addressing modes for the "X:<ea_MM>,DDDDD" instruction.
       MM     addressing mode
       --     ---------------
       00     (Rn)+  or    (SP)+
       01     (Rn)+N  or    (SP)+N
       10     (Rn)-  or    (SP)-
       11     (Rn)   or    (SP)       (LEA cannot use this combination)

m:  specifies addressing modes of "P1DALU" and "P2DALU"
       m      addressing mode
       -      ---------------
       0      (Rn)+
       1      (Rn)+N

W:
       W      move direction for memory moves
       -      -------------------------------
       0      register -> memory
       1      memory   -> register

w:     w      DALU result
       -      -----------
       0      written back to memory   (not allowed for CMP or SUB instrs)
       1      remains in register

Immediates and Absolute Address Instruction Field Definitions:
==============================================================

AAA:
       Upper 3 address bits for JMP, Jcc, and JSR instructions.

BBBBBBB:
```

7-bit signed integer. For #xx,HHHH and DALU #xx,F instructions.

BBBBBB:
6-bit unsigned integer. For DO/REP #xx instruction.

AAAAAA:
6-bit positive offset for X:(R2+xx) addressing mode.
Allows positive offsets: 0 to 63

aaaaaa:
6-bit negative offset for X:(SP-xx) addressing mode.
Allows negative offsets: -1 to -64

Aaaaaaa:
7-bit offset for MOVE, DALU & Bitfield using X:(SP-#xx), X:(R2+#xx)
and Bcc <aa> instructions:

```
   A = 0   =>   X:(R2+#xx)   allows positive offsets:  0 to  63
   A = 1   =>   X:(SP-#xx)   allows negative offsets: -1 to -64
```

For Bcc, "A" specifies the sign-extension.
RESTRICTION: Aaaaaaa must never be all zeros for the Bcc instruction.

Ppppppp:
7-bit absolute address for MOVE, DALU, & Bitfield on X:<pp> instr
It is sign-extended to allow access to both the peripherals and
the 1st 64 locations in X-memory.

Other Instruction Field Definitions:

Z: specifies the parallel moves of the address pointers in a Tcc instruction.

Z	move	
0	R0->R0	(i.e., no transfer occurs in the AGU unit)
1	R0->R1	(AGU transfers R0 register to R1 if condition true)

For the case where Z=0, the assembler will not look for a field
such as "teq x0,a r0,r0". Instead, the AGU register transfer
will be suppressed, such as in ""teq x0,a".

E:

E	instruction
0	DO
1	REP

tt:

tt	instruction
00	STOP
01	WAIT
10	SWI
11	ILLEGAL

BITFIELD:
UUU: specifies bitfield/branch-on-bit instructions

UUU	operations
000	BFCLR
001	BFSET
010	BFCHG
011	MOVE (used by "move #iiii,<ea>")
100	BFTSTL
110	BFTSTH
101	BRCLR (modifies carry bit)
111	BRSET (modifies carry bit)
0xx	last word = iiiiiiiiiiiiiiii
1x0	last word = iiiiiiiiiiiiiiii

```
        1x1    last word = iiiiiiiiUAaaaaaa

        (note: this is the 3rd word, not 2nd, for BF/BR #xxxx,X:xxxx)

                  iiiiiiiiiiiiiiii = 16-bit immed mask
                  iiiiiiii      = 8-bit  immed mask for upper or lower byte
                  U = 1           selects upper byte
                  U = 0           selects lower byte
                  Aaaaaaa       = 7-bit relative branch field

                  Note: UAaaaaaa is not available to the BFTSTH, BFTSTL instrs

        The ANDC, ORC, EORC, and NOTC are instructions which fall directly
        onto the bitfield instructions.  They are mapped as follows:

              ANDC is identical to a BFCLR with the mask inverted
              ORC  is identical to a BFSET (mask not inverted)
              EORC is identical to a BFCHG (mask not inverted)
              NOTC is identical to a BFCHG with the mask set to $FFFF

CC-C: ()
       Specifies conditions for the Tcc instructions:
       (in this case, "CC" falls onto C10 of CCCC, "C" falls onto C2, C3 is "0")
        CC-C   condition
        ----   ---------
        00 0   cc
        01 0   cs
        10 0   ne
        11 0   eq

        00 1   ge
        01 1   lt
        10 1   gt
        11 1   le

CCCC: ()
       Specifies conditions for the Jcc, JScc, and Bcc instructions

        CCCC   condition - for encode7
        ----   ---------
        0000   cc (same as "hs", unsigned higher or same)
        0001   cs (same as "lo", unsigned lower)
        0010   ne
        0011   eq
        0100   ge
        0101   lt
        0110   gt
        0111   le

        10**   ALWAYS TRUE condition            (PLAs decode this)

        1001   ALWAYS - JMP, BRA, JSR     (value used by assembler)
        1011   (reserved -could be used for delayed)
        1010   (reserved)
        1000   (reserved)
        1100   hi (unsigned higher)
        1101   ls (unsigned lower or same)
        1110   nn
        1111   nr

Unusual Instruction Encodings:
==============================
 Encoding of "ADD fff,X:<aa>" and "ADD fff,X:(sp-xx)":
        There is an unusual trick used to encode these two instructions.
        What is so unusual is that the first word of the two word
        "ADD/SUB/CMP fff,X:<aa>" instruction is identical to the one
        word encoding of the "ADD/SUB/CMP X:<aa>,fff" instruction.
        It is also true the first word of the two word
        "ADD/SUB/CMP fff,X:(sp-xx)" instruction is identical to the one
        word encoding of the "ADD/SUB/CMP X:(sp-xx),fff" instruction.

        What makes these instructions differ is the encoding of the instruction
        immediately following the first word.  The rules are listed below.

        Encoding Rules:

           ADD X:<aa>,fff:
              - 1st word - Simply uses the one word encoding for ADD X:<aa>,fff
              - 2nd word - Any valid DSP56800 instruction, which by definition
                         will not be the following reserved hex value: $E042.
                         Note that this value is reserved in the DSP56800
                         bit encoding map.

           ADD X:(SP-xx),fff:
              - 1st word - Simply uses the one word encoding for
                         ADD X:(SP-xx),fff
              - 2nd word - Any valid DSP56800 instruction, which by definition
                         will not be the following reserved hex value: $E042.
                         Note that this value is reserved in the DSP56800
                         bit encoding map.

           ADD X:xxxx,fff:
              - 1st word - 1st word of encoding uses ADD X:xxxx,fff
                         with the "w" bit set to "1"
              - 2nd word - second word of encoding contains the 16-bit
                         absolute address

           ADD fff,X:<aa>:
              - 1st word - 1st word of this instruction uses the one word
                         encoding for the ADD X:<aa>,fff instruction.
              - 2nd word - 2nd word of this instruction is simply set to $E042.

           ADD fff,X:(SP-xx):
              - 1st word - 1st word of this instruction uses the one word
                         encoding for the ADD X:(SP-xx),fff instruction.
              - 2nd word - 2nd word of this instruction is simply set to $E042.

           ADD fff,X:xxxx:
              - 1st word - 1st word of encoding uses ADD X:xxxx,fff
                         with the "w" bit set to "0"
              - 2nd word - second word of the instruction contains the 16-bit
                         absolute address

        Thus, the presence of the hex value $E042 in the instruction
        immediately after a "ADD X:<aa>,fff" or "ADD X:(sp-xx),fff"
        indicates that the instruction is really an "ADD fff,X:<aa>" or
        "ADD fff,X:(sp-xx)" instruction.  These later two instructions
        encode as two word instructions using the technique described above.

        Note that this encoding (where the destination is a memory
        location) is NOT allowed for the SUB or CMP instructions.
        It is only allowed for the ADD instruction.

Encoding of LEA:
        There is a trick used for encoding the LEA instruction.  The trick
        is used in several different places within the opcode map and is
        simply this - anytime a MOVE instruction uses "NOREG" (located in the
        HHHH or DDDDD field) as a source register, the instruction is no longer
        interpreted as a MOVE instruction.  Instead it operates as an LEA
        instruction.  Thus, the syntax for the instruction available to the
        user is "LEA", but the actual bit encoding uses the MOVE instruction
        where the source register is "NOREG":

           DSP56800 Instruction        Encoded As:
           --------------------        -----------
            LEA  (Rn)+          =>   MOVE  NOREG,X:(Rn)+
            LEA  (Rn)-         =>   MOVE  NOREG,X:(Rn)-
            LEA  (Rn)+N        =>   MOVE  NOREG,X:(Rn)+N
            LEA  (R2+xx)       =>   MOVE  NOREG,X:(R2+xx)
            LEA  (Rn+xxxx)     =>   MOVE  NOREG,X:(Rn+xxxx)

            LEA  (SP)+         =>   MOVE  NOREG,X:(SP)+
            LEA  (SP)-         =>   MOVE  NOREG,X:(SP)-
            LEA  (SP)+N        =>   MOVE  NOREG,X:(SP)+N
            LEA  (SP-xx)       =>   MOVE  NOREG,X:(SP-xx)
            LEA  (SP+xxxx)     =>   MOVE  NOREG,X:(SP+xxxx)

        CAREFUL: LEA must NOT write to a memory location!
        NOTE:   LEA not allowed for (Rn) or (SP).
```

Encoding of TSTW:

There is a trick used for encoding the TSTW instruction. The trick is used in several different places within the opcode map and is simply this - anytime a MOVE instruction uses "NOREG" (located in the HHHH or DDDDD field) as a dest register, the instruction is no longer interpreted as a MOVE instruction. Instead it operates as a TSTW instruction. Thus, the syntax for the instruction available to the user is "TSTW", but the actual bit encoding uses the MOVE instruction where the destination register is "NOREG":

```
   DSP56800 Instruction        Encoded As:
   --------------------        -----------
    TSTW X:<aa>        =>   MOVE  X:<aa>,NOREG
    TSTW X:<pp>        =>   MOVE  X:<pp>,NOREG
    TSTW X:xxxx        =>   MOVE  X:xxxx,NOREG
    TSTW X:(Rn)        =>   MOVE  X:(Rn),NOREG
    TSTW X:(Rn)+       =>   MOVE  X:(Rn)+,NOREG
    TSTW X:(Rn)-       =>   MOVE  X:(Rn)-,NOREG
    TSTW X:(Rn)+N      =>   MOVE  X:(Rn)+N,NOREG
    TSTW X:(Rn+N)      =>   MOVE  X:(Rn+N),NOREG
    TSTW X:(Rn+xxxx)   =>   MOVE  X:(Rn+xxxx),NOREG
    TSTW X:(R2+xx)     =>   MOVE  X:(R2+xx),NOREG
    TSTW X:(SP)        =>   MOVE  X:(SP),NOREG
    TSTW X:(SP)+       =>   MOVE  X:(SP)+,NOREG
    TSTW X:(SP)-       =>   MOVE  X:(SP)-,NOREG
    TSTW X:(SP)+N      =>   MOVE  X:(SP)+N,NOREG
    TSTW X:(SP+N)      =>   MOVE  X:(SP+N),NOREG
    TSTW X:(SP+xxxx)   =>   MOVE  X:(SP+xxxx),NOREG
    TSTW X:(SP-xx)     =>   MOVE  X:(SP-xx),NOREG
    TSTW <register>    =>   MOVE  ddddd,NOREG

NOTE: TSTW (Rn)- is not encoded in this manner, but instead
      has its own encoding allocated to it.

NOTE: TSTW HWS is NOT allowed.  All other on-chip registers
      are allowed.

IMPORTANT NOTE: TSTW can be done on any other instruction which
   allows a move to NOREG.  Note this doesn't make sense for LEA.

NOTE: TSTW F (operates on saturated 16 bits) differs
      from TST F (operates on full 36/32 bit accumulator)

NOTE: TSTW P:() is NOT allowed.
```

Encoding of POP:

The encoding of the POP follows the simple rules below.

```
   DSP56800 Instruction        Encoded As:
   --------------------        -----------
    POP  <reg>         =>   MOVE X:(SP)-,<reg>
    POP                =>   LEA  (SP)-
```

In the first case, a register is explicitely mentioned, whereas in

```
        the second case, no register is specified, i.e., just removing a value
        from the stack.

        NOTE: There is no PUSH instruction, but it is easy to write
              a simple two word macro for PUSH.

Encoding of CLR:
        The encoding for a CLR on anything other than A or B
        should encode into the following: "move #0,<reg>".
        Allows the following instructions to be recognized by the assembler:
           CLR DD         (DD = x0,y0,y1)
           CLR F1         (F1 = a1,b1)
           CLR RR         (DD = r0,r1,r2,r3)
           CLR N
        Note that no parallel move is allowed with these.
        Note also that CLR F sets the condition codes,
        whereas CLR on DD, F1, RR, or N does NOT set the condition codes.

Encoding of ENDDO:
        The ENDDO instruction will be encoded as "MOV HWS,NOREG".

Encoding of the Tcc Instruction:
--------------------------------
The Tcc instruction is somewhat difficult to understand because it's encoding
overlays the encodings of some Data ALU instructions when Bit 2 of the opcode
is a "1".  It is overlayed obviously so that for a particular bit pattern,
there is only one unique instruction present.  Reference to this can be seen
with the "<<Tc>>" entry found within Charts 3 and 4 below.  Use the definition

       "0110 11CC FJJJ 01CZ Tcc  JJJ,F  [R0->R1]"

to encode this instruction.

============================================================================
============================================================================

Restrictions:
-------------
   - The HWS register cannot be specified as the loop count for a DO or
     REP instruction.  Likewise, no bitfield operations (BFTSTH, BFTSTL,
     BFSET, BFCLR, BFCHG, BRSET, BRCLR) can operate on the HWS register.
     Note, however, that all other instructions which access ddddd, including
     "move #xxxx,HWS" and TSTW, can operate on the HWS register.
   - The following registers cannot be specified as the loop count for a DO or
     REP instruction - HWS, M01, SR, OMR.
   - The "lea" instruction does NOT allow the (Rn) addressing mode, i.e.,
     it only allows (Rn)+, (Rn)-, (Rn)+N, (Rn+xxxx), (R2+xx), and (SP-xx)
   - Cannot do a bitfield set/clr/change on "ND" register, i.e., the bitfield
     instruction cannot be immediately followed by an instruction which uses
     the "N" register in an addressing mode.
             bfclr  #$1234,n
             move   x:(r0+n),x0         ; illegal - needs one NOP
     Special care is necessary in hardware loops, where the instruction at
     LA is followed by the instruction at the top of the loop as well as the
     instruction at LA+1.
   - Cannot move a long immediate value to the "ND" register.  This is because
     the long immediate move is implemented similar to the bitfield instrs.
       move   #$1234,n           ; long immediate
       move   x:(r0+n),x0        ; ILLEGAL - needs one NOP

       move   #$4,n              ; short immediate, uses ND register
       move   x:(r0+n),x0        ; ALLOWED since uses short immediate
   - The value "0000000" is not allowed for Bcc.
     In addition, this same value is not allowed as the relative offset
     for a BRSET or BRCLR instruction.
   - The value "0" is not allowed for the DO #xx instruction.
     If this case is encountered by the assembler, it should not be accepted.
   - Jumps to LA and LA-1 of a hardware loop are not allowed.  This also
```

```
     applies to the BRSET and BRCLR instructions.
   - A NORM instruction cannot be immediately followed by an instruction
     which uses the Address ALU register modified by the NORM instruction
     in an addressing mode.
             norm   r0,a
             move   x:(r0)+,x0          ; illegal - needs one NOP
     Special care is necessary in hardware loops, where the instruction at
     LA is followed by the instruction at the top of the loop as well as the
     instruction at LA+1.
   - Only positive values less than 8192 can be moved to the LC register.
   - Cannot REP on any multiword instruction or any instruction which
     performs a P:() memory move.
   - Cannot REP on any instruction not allowed on the DSP56100.
   - IF a MOVE or bitfield instruction changes the value in R0-R3 or SP,
     then the contents of the register are not available for use until the
     second following instruction, i.e., the immediately following instruction
     should not use the modified register to access X memory or update an
     address.  This restriction does NOT apply to the N register or the
     (Rn+xxxx) addressing mode as discussed below.
   - For the case of nested looping, it is required that there are at least
     two instruction cycles after the pop of the LC and LA registers before
     the instruction at LA for the outer loop.
   - A hardware DO loop can never cross a 64K program memory boundary, i.e.,
     the DO instruction as well as the instruction at LA must both reside
     in the same 64K program memory page.
   - Jcc, JMP, Bcc, BRA, JSR, BRSET or BRCLR instructions are not allowed in
     the last two locations of a hardware do loop, i.e., at LA, and LA-1.
     This also means that a two word Jcc, JMP, or JSR instruction may not have
     its first word at LA-2, since its second word would then be at LA-1, which
     is not allowed.

Restrictions Removed:
---------------------
   - The following instruction sequence is NOW ALLOWED:
             move   <>,lc        ; move anything to LC reg
             do     lc,label            ; immediately followed by DO
     This was not allowed on the 56100 family due to its internal pipeline.
   - An AALU pipeline NOP is not required in the following case:
             move   <>,Rn        ; same Rn as in following instr
             move   X:(Rn+xxxx),<>      ; OK, no NOP required!

             move   <>,Rn        ; same Rn as in following instr
             move   <>,X:(Rn+xxxx)      ; OK, no NOP required!

     In this case, there will NOT be an extra instruction cycle inserted
     because any move with the X:(Rn+xxxx) or X:(SP+xxxx) addressing mode
     is already a 3 Icyc instruction.
   - An AALU pipeline NOP is not required in the following case:
             move   <>,Rn               ; same Rn as in following instr
             lea    (Rn+xxxx)           ; OK, no NOP required!

     In this case, there will NOT be an extra instruction cycle inserted
     because any lea with the (Rn+xxxx) or (SP+xxxx) addressing mode
     is already a 2 Icyc instruction.
   - An AALU pipeline NOP is not required in the following case:
             move   <>,N
             move   X:(Rn+N),<>         ; OK, no NOP required!

             move   <>,N
             move   <>,X:(Rn+N)         ; OK, no NOP required!

             move   <>,N
             move   <>,X:(Rn)+N         ; OK, no NOP required!

             move   <>,N
             move   X:(Rn)+N,<>         ; OK, no NOP required!

     In this case, there WILL be an extra instruction cycle inserted
```

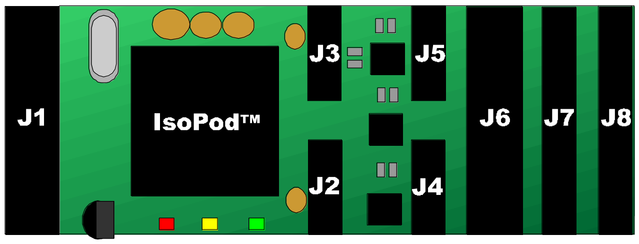
and the assembler will use the ND register, not the N register.

185. APPENDIX: IsoPod™ V1 HARDWARE REFERENCE

186. *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

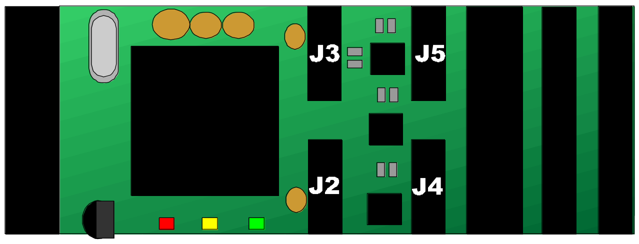


187. J1 GPIO

+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).

188. J3 IO/SPI V1

+3V	1	2	GND
PD0	3	4	PE4/SCLK
PD1	5	6	PE5/MOSI
PD2	7	8	PE6/MISO
PD3	9	10	PE7SS'

189. J2 JTAG V1

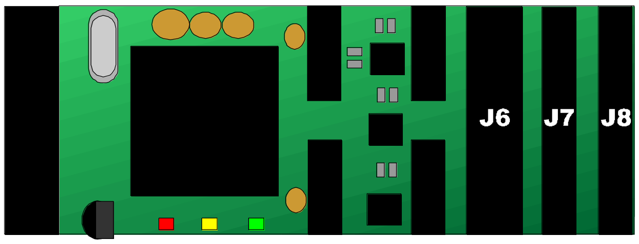
+3V	1	2	GND
TDI	3	4	GND
TDO	5	6	TMS
TCK	7	8	DE
RESET'	9	10	TRST

190. 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.

191. 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).

192. 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

193. J7 Motor Encoder x 2 V1

+5V	1	2	FAULTA0
GND	3	4	FAULTA1
PH ASEA0	5	6	FAULTA2
PHASE B0	7	8	ISA0
INDEX0	9	10	ISA1
HOME0	11	12	ISA2
+5V	13	14	FAULTB0
GND	15	16	FAULTB1
PHASEA1	17	18	FAULTB2
PHASEB1	19	20	ISB0
INDEX1	21	22	ISB1
HOME1	23	24	ISB2

194. 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'

195. 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.

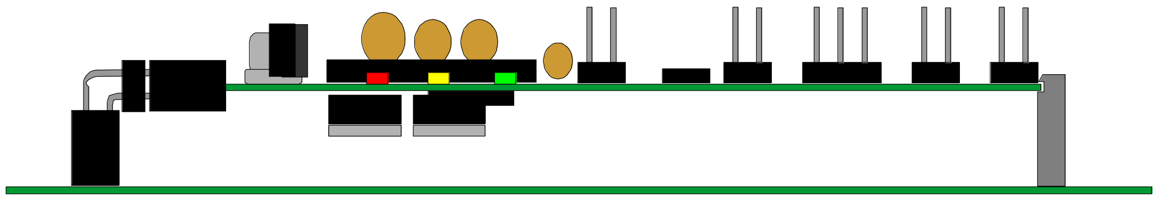
Port line TD0 controls the RS-232 transmitter shutdown.

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

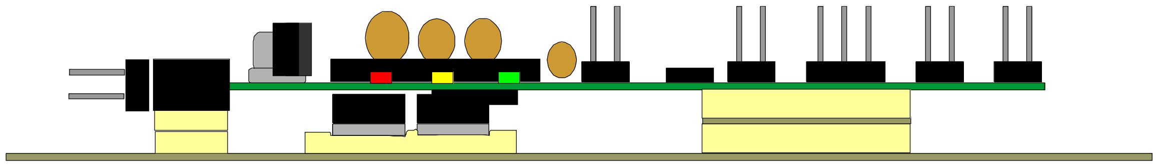
196. BOARD MOUNTING V1

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

197. J1 and supporting clip:

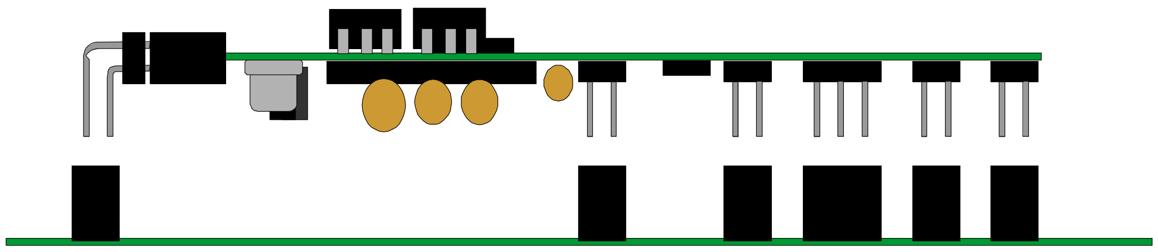


198. Double sided sticky tape:



199. Inversion and insertion:

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



200. Cable or adapter:

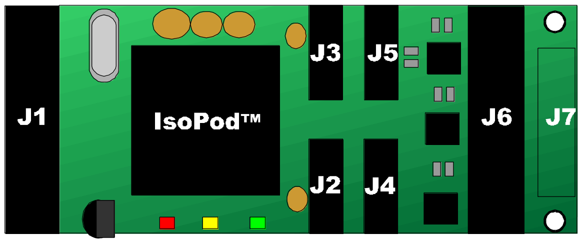
An IDC cable with an IDC male connector can, or an IDC female used with an intermediate double male header, can be ribbon cabled to a similar IDC 24-pin socket header and plugged into an existing stamp-type socket. NMI also manufactures a level, and a right angle adapter for the same purpose.

201. APPENDIX: IsoPod™ V2 HARDWARE REFERENCE

202. 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

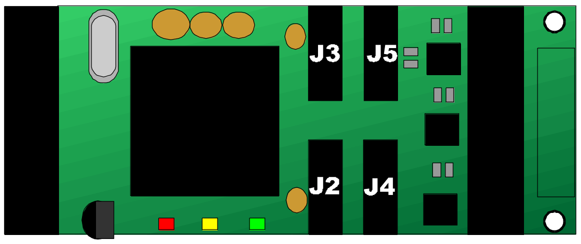


203. J1 GPIO V2

+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).

204. J3 A/D V2

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

206. J5 IO/SPI V2

+5V	1	2	GND
+3V	3	4	PE4/SCLK
RST0’	5	6	PE5/MOSI
PE2	7	8	PE6/MISO
PE3	9	10	PE7/SS’

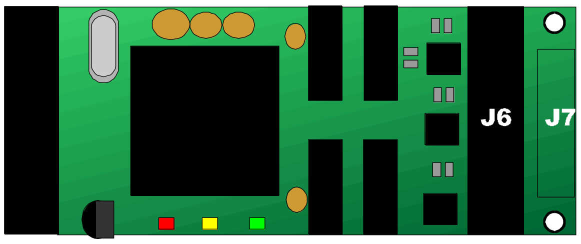
205. J2 JTAG V2

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

207. 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 by default. RS-422/485 is optional



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

208. J6 PWM, Motor Encoder, Timers V2

1	PWMA0	2	+5V	3	+3V
4	PWMA1	5	GND	6	GND
7	PWMA2	8	PHASEA0/TA0	9	TMRC0
10	PWMA3	11	PHASEB0/TA1	12	TMRC1
13	PWMA4	14	INDEX0/TA2	15	IRQA
16	PWMA5	17	HOME0/TA3	18	IRQB
19	PWMB0	20	+5V	21	+3V
22	PWMB1	23	GND	24	GND
25	PWMB2	26	PHASEA1/TB0	27	TMRD0
28	PWMB3	29	PHASEB1/TB1	30	TMRD1
31	PWMB4	32	INDEX1/TB2	33	TMRD2
34	PWMB5	35	HOME1	36	TMRD3

209. 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.

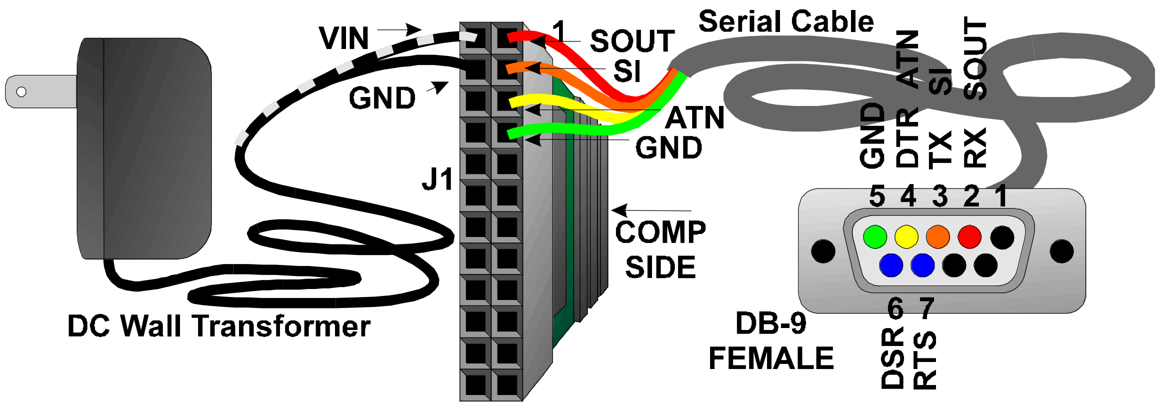
210. Instructions for Wiring a Serial Cable V1 & V2

211. 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

212. 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

213. 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.

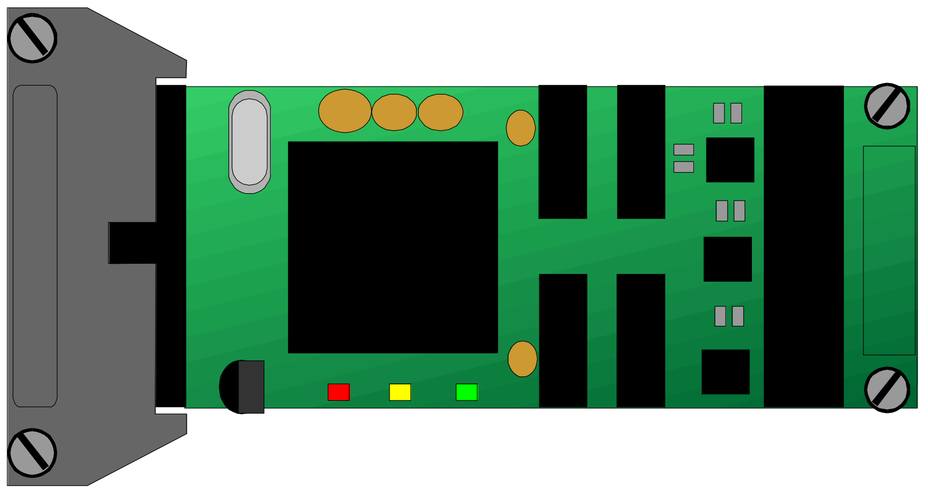
Port line PD0 controls the GREEN LED. The built in pull up in the AC05 makes the LED come on, if the port line is inactive.

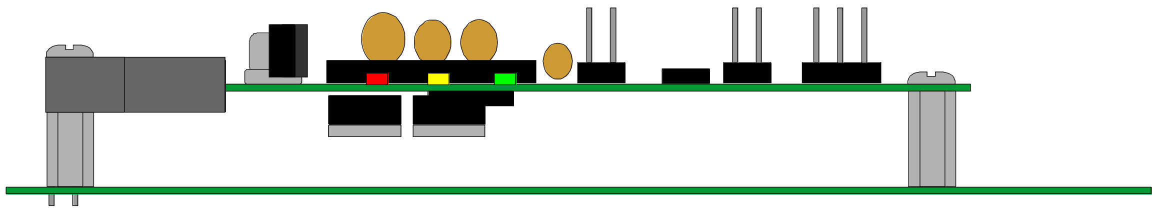
Port line PE2 controls the CAN transceiver mode, high-speed mode or silent mode. A pull down resistor normally selects high-speed mode, if the port line is inactive. In the silent mode, the transmitter is disabled. All other IC functions continue to operate. The silent mode is selected by connecting 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.

214. *BOARD MOUNTING V2*

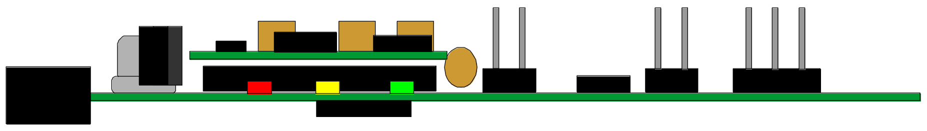
Two mounting holes are provided on the IsoPod™ Board V2:

215. J1 Wall Header and supporting standoffs:





216. V2 Switching Regulator (SR) option:



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

217. 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

218. 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.

219. 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.

220. SUPPORTING SOFTWARE

221. 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.LNK

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
```

222. 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.

223. REFERENCE

224. *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
033	21	!
034	22	"
035	23	#
036	24	$
037	25	%
038	26	&
039	27	'
040	28	(
041	29	)
042	2A	*
043	2B	+
044	2C	,
045	2D	-
046	2E	.
047	2F	/
048	30	0
049	31	1
050	32	2
051	33	3
052	34	4
053	35	5
054	36	6
055	37	7
056	38	8
057	39	9
058	3A	:
059	3B	;
060	3C	<
061	3D	=
062	3E	>
063	3F	?
064	40	@
065	41	A
066	42	B
067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4A	J
075	4B	K
076	4C	L
077	4D	M
078	4E	N
079	4F	O
080	50	P
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5A	Z
091	5B	[
092	5C	\
093	5D	]
094	5E	^
095	5F	_
096	60	`
097	61	a
098	62	b
099	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	I
106	6A	J
107	6B	K
108	6C	L
109	6D	M
110	6E	N
111	6F	O
112	70	P
113	71	Q
114	72	R
115	73	S
116	74	T
117	75	U
118	76	V
119	77	W
120	78	X
121	79	Y
122	7A	Z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	DEL

225. 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

226. ISOMAX GLOSSARY

Stack comments use the following notation:

n	a signed 16-bit value, -32768..+32767.
u	an unsigned 16-bit value, 0..65535.
+n	a signed, positive 16-bit value, 0..+32767.
w	a generic16-bit value.
16b	a generic 16-bit value.
addr	an address (16 bits).
c	a character. (Note: stored as 16 bits on the IsoPod™)
8b	a generic 8-bit value. (Note: stored as 16 bits on the IsoPod™)
d	a signed 32-bit value, -2,147,483,648..+2,147,483,647.
ud	an unsigned 32-bit value, 0..4,294,967,295.
wd	a generic 32-bit value.
32b	a generic 32-bit value.
r	a floating-point (real) value.
flag	a logical flag, zero = false, -1 (all ones) = true.

Values on the stack before and after execution of a word are given as follows:

(before --- after)	normal integer data stack
(F: before --- after)	floating-point data stack
(C: before --- after)	compile-time behavior of the integer data stack.

Stack comments in italics also refer to compile-time behavior.

227. Integer Arithmetic

Word	**Stack Effect**	**Description**
*	(w1 w2 --- w3)	Multiplies w2 by w1 and leaves the product w3 on the stack.
*/	(n1 n2 n3 --- n4)	Multiplies n2 by n1 and divides the product by n3. The quotient, n4 is placed on the stack.
*/MOD	(n1 n2 n3 -- n4 n5)	n1 is multiplied by n2 producing a product which is divided by n3. The remainder, n4 and the quotient, n5 are then placed on the stack.
+	(w1 w2 --- w3)	Adds w2 and w1 then leaves the sum, w3 on the stack.
+!	(w1 addr ---)	Adds w1 to the value at addr then stores the sum at addr replacing its previous value.
-	(w1 w2 --- w3)	Subtracts w2 from w1 and leaves the result, w3 on the stack.

/	(n1 n2 --- n3)	Divides n1 by n2 and leaves the quotient n3 on the stack.
/MOD	(n1 n2 --- n3 n4)	Divides n1 by n2 then leaves on the stack the remainder n3 and the quotient n4.
1+	(w1 --- w2)	Adds 1 to w1 then leaves the sum, w2 on the stack.
1+!	(addr ---)	Adds one to the value at addr and stores the result at addr.
1-	(w1 --- w2)	Subtract 1 from w1 then leaves the difference, w2 on the stack.
1-!	(addr ---)	Subtracts one from the value at addr and stores the result at addr.
2*	(w1 --- w2)	Multiplies w1 by 2 to give w2.
2+	(w1 --- w2)	Adds two to w1 and leaves the sum, w2 on the stack.
2-	(w1 --- w2)	Subtracts two from w1 and leaves the result, w2 on the stack.
2/	(n1 --- n2)	Divides n1 by 2, giving n2 as the result.
><	(8b1/8b2 --- 8b2/8b1)	Swaps the upper and lower bytes of the value on the stack.
ABS	(n --- u)	Leaves on the stack the absolute value, u of n.
MAX	(n1 n2 --- n3)	Leaves the greater of n1 and n2 as n3.
MIN	(n1 n2 --- n3)	Leaves the lesser of n1 and n2 as n3.
MOD	(n1 n2 --- n3)	Divides n1 by n2 and leaves the remainder n3.
NEGATE	(n1 --- n2)	Leaves the two's complement n2 of n1.
UM*	(u1 u2 ---ud)	Multiplies u1 and u2 returning the double length product ud.
UM/MOD	(ud u1 --- u2 u3)	Divides the double length unsigned number ud by u1 and returns the single length remainder u2 and the single length quotient u3.

228. Logical and Comparison

Word	Stack Effect	Description
0<	(n --- flag)	Leaves a true flag if n is less than zero.
0=	(w --- flag)	Leaves a true flag if w is equal to zero.
0>	(n --- flag)	Leaves a true flag if n is greater than zero.
<	(n1 n2 --- flag)	Leaves a true flag on stack if n1 is less than n2.
=	(w1 w2 --- flag)	Returns a true flag if w1 is equal to w2.
>	(n1 n2 --- flag)	Returns a true flag if n1 is greater than n2.
AND	(16b1 16b2 --- 16b3)	Leaves the bitwise logical AND of 16b1 and 16b2 as 16b3.
CLEAR-BITS		Clears bits at addr corresponding to 1s in mask b.
INVERT	(16b1 --- 16b2)	Leaves the one's complement 16b2 of 16b1.
NOT	(flag1 --- flag2)	Leaves the logical inverse flag2 of flag1. flag2 is false if flag1 was true, and vice versa.
OR	(16b1 16b2 ---	Leaves the inclusive-or 16b3 of 16b1 an 16b2.

	16b3)	
SET-BITS	(b addr ---)	Sets bits at addr corresponding to 1s in mask b.
TOGGLE-BITS	(b addr ---)	Toggles bits at addr corresponding to 1s in mask b.
U<	(u1 u2 ---flag)	Returns a true flag if u1 is less then u2.
XOR	(16b1 16b2 --- 16b3)	Performs a bit-by-bit exclusive or of 16b1 with 16b2 to give 16b3.

229. Double-Precision Operations

Word	**Stack Effect**	**Description**
2CONSTANT <name>	(32b ---)	Creates a double length constant for a <name>. When <name> is executed, 32b is left on the stack.
2DROP	(32b ---)	Removes 32b from the stack.
2DUP	(32b --- 32b 32b)	Duplicates 32b.
2OVER	(32b1 32b2 --- 32b1 32b2 32b3)	32b3 is a copy of 32b1
2ROT	(32b1 32b2 32b3 --- 32b2 32b3 32b1)	Rotates 32b1 to the top of the stack.
2SWAP	(32b1 32b2 --- 32b2 32b1)	Swaps 32b1 and 32b2 on the stack.
2VARIABLE <name>	(---)	Creates double-length variable for <name>. when <name> is executed, its parameter field address is placed on the stack.
D*	(d1 d2 --- d3)	Multiplies d1 by d2 and leaves the product d3 on the stack.
D+	(wd1 wd2 --- wd3)	Adds wd1 and wd2 and leaves the result, wd3 on stack.
D-	(wd1 wd2 --- wd3)	Subtracts wd2 from wd1 and returns the dif- ference wd3.
D/	(d1 d2 --- d3)	Divides d1 by d2 and leaves the quotient d3 on the stack.
D0=	(wd --- flag)	Returns a true flag if wd is equal to zero.
D2/	(d1 --- d2)	Divides d1 by 2 and gives quotient d2.
D<	(d1 d2 --- flag)	Leaves a true flag if d1 is less than d2; otherwise leaves a false flag.
D=	(wd1 wd2 --- flag)	Returns a true flag if wd1 is equal to wd2.
DABS	(d --- ud)	Returns the absolute value of d as ud.
DCONSTANT <name>	(32b ---)	Creates a double length constant for a <name>. When <name> is executed, 32b is left on the stack. Same as 2CONSTANT.

DDROP	(32b ---)	Removes 32b from the stack. Same as 2DROP.
DDUP	(32b --- 32b 32b)	Duplicates 32b. Same as 2DUP.
DMAX	(d1 d2 --- d3)	Returns d3 as the greater of d1 or d2.
DMIN	(d1 d2 --- d3)	Returns d3 as the lesser of d1 or d2.
DMOD	(d1 d2 --- d3)	Divides d1 by d2 and leaves the remainder d3.
DNEGATE	(d1 --- d2)	Leaves the two's complement d2 of d1.
DOVER	(32b1 32b2 --- 32b1 32b2 32b3)	32b3 is a copy of 32b1. Same as 2OVER.
DROT	(32b1 32b2 32b3 --- 32b2 32b3 32b1)	Rotates 32b1 to the top of the stack. Same as 2ROT.
DSWAP	(32b1 32b2 --- 32b2 32b1)	Swaps 32b1 and 32b2 on the stack. Same as 2SWAP.
DU<	(ud1 ud2 --- flag)	Returns a true flag if ud1 is less than ud2.
DVARIABLE <name>	(---)	Creates double-length variable for <name>. when <name> is executed, its parameter field address is placed on the stack. Same as 2VARIABLE.
S->D	(n --- d)	Sign extend single number to double number.

230. Floating-point Operations

Word	**Stack Effect**	**Description**
2**X	(F: r1 -- r2)	Raise 2 to the r1 power giving r2.
D>F	(d --) (F: -- r)	R is the floating-point equivalent of d.
e	(F: -- r1)	Put natural value e (=2.718282) on the floating-point stack as r1.
F!	(addr --) (F:r --)	Store r at addr.
F*	(F:r1 r2 -- r3)	Multiply r1 by r2 giving r3.
F**	(F:r1 r2 -- r3)	Raise r1 to the r2 power giving r3.
F+	(F:r1 r2 -- r3)	Add r1 to r2, giving r3.
F,	(F:r --)	Store r as a floating-point number in the next available dictionary location.
F-	(F:r1 r2 -- r3)	Subtract r2 from r1, giving r3.
F/	(F:r1 r2 -- r3)	Divide r1 by r2, giving r3.
F0<	(F:r --) (-- flag)	flag is true if r is less than zero.
F0=	(F:r --) (-- flag)	flag is true if r is equal to zero.
F2*	(F:r1 -- r2)	Multiply r1 by 2 giving r2.
F2/	(F:r1 -- r2)	Divide r1 by 2 giving r2.
F<	(F:r1 r2 --)(-- flag)	flag is true if r1 is less than r2.
F>D	(F:r --)(-- d)	Convert r to d.
F@	(addr --)(F: -- r)	r is the value stored at addr.

FABS	(F:r1 -- r2)	R2 is the absolute value of r1.
FALOG	(F:r1 -- r2)	Raise 10 to the power r1, giving r2.
FATAN	(F:r1 -- r2)	R2 is the principal radian whose tangent is r1.
FATAN2	(F:r1 r2 -- r3)	R3 is the radian angle whose tangent is r1/r2.
FCONSTANT <name>	(F:r --)	Define a constant <name> with value r.
FCOS	(F:r1 -- r2)	r2 is the cosine of the radian angle r1.
FDEPTH	(-- +n)	+n is the number of values contained on separate floating point stack.
FDROP	(F:r--)	Remove r from the floating-point stack.
FDUP	(F:r -- r r)	Duplicate r.
FEXP	(F:r1 -- r2)	Raise e to the power r1, giving r2.
FLN	(F:r1 -- r2)	R2 is the natural logarithm of r1.
FLOAT+	(addr1 -- addr2)	Add the size of a floating-point value to addr1.
FLOATS	(n1 -- n2)	n2 is the size, in bytes, of n1 floating-point numbers.
FLOG	(F:r1 -- r2)	R2 is the base 10 logarithm of r1.
FLOOR	(F:r1 -- r2)	Round r1 using the "round to negative infinity" rule, giving r2.
FMAX	(F:r1 r2 -- r3)	r3 is the maximum of r1 and r2.
FMIN	(F:r1 r2 -- r3)	r3 is the minimum of r2 and r3.
FNEGATE	(F:r1 -- r2)	r2 is the negation of r1.
FNIP	(F:r1 r2 -- r2)	Remove second number down from floating-point stack.
FOVER	(F:r1 r2 -- r1 r2 r1)	Place a copy of r1 on top of the floating-point stack.
FROUND	(F:r1 -- r2)	Round r1 using the ";round to even"; rule, giving r2.
FSIN	(F:r1 -- r2)	R2 is the sine of the radian angle r1.
FSQRT	(F:r1 -- r2)	R2 is the square root of r1.
FSWAP	(F:r1 r2 -- r2 r1)	Exchange the top two floating-point stack items.
FTAN	(F:r1 -- r2)	R2 is the tangent of the radian angle r1.
FVARIABLE <name>	(--)	Create a floating-point variable <name>. Reserve data memory in the dictionary sufficient to hold a floating-point value.
LOG2	(F:r1 -- r2)	R2 is the base 2 logarithm of r1.
ODD-POLY	(F: -- r1)(addr --)	Evaluate odd-polynomial giving r1.
PI	(F: -- r1)	Put the numerical value of pi on the floating- point stack as r1.
POLY	(F: -- r1)(addr --)	Evaluate polynomial giving r1.
S>F	(n--)(F: -- r)	R is the floating-point equivalent of n.
SF!	(addr --)(F:r --)	Store the floating point number r as a 32 bit IEEE single precision number at addr.
SF@	(addr --)(F: -- r)	Fetch the 32-bit IEEE single precision number stored at addr to the floating-point stack as r in the internal representation.

231. Stack Operations

Word	**Stack Effect**	**Description**
-ROLL	(n ---)	Removes the value on the top of stack and inserts it into the nth place from the top of stack.
>R	(16b ---)	Removes 16b from user stack and place it onto return stack.
?DUP	(16b --- 16b 16b), (0 --- 0)	Duplicates 16b if it is a non-zero.
DEPTH	(--- +n)	Returns count +n of numbers on the data stack.
DROP	(16b ---)	Removes 16b from the data stack.
DUP	(16b --- 16b 16b)	Duplicates 16b.
OVER	(16b1 16b2 --- 16b1 16b2 16b3)	16b3 is a copy of 16b1.
PICK	(+n --- 16b)	Copies the data stack's +nth item onto the top.
R>	(--- 16b)	16b is removed from the return stack and placed onto the data stack.
R@	(--- 16b)	16b is a copy of the top of the return stack.
ROLL	(+n ---)	Removes the stack's nth item and places it onto the top of stack.
ROT	(16b1 16b2 16b3 --- 16b2 16b3 16b1)	Rotates 16b1 to the top of the stack.
RP!	(--)	Initializes the bottom of the return stack.
RP@	(-- addr)	addr is the address of the top of the return stack just before RP@ was executed.
SP!	(--)	Initializes the bottom of the parameter stack.
SP@	(--- addr)	addr is the address of the top of the parameter stack just before SP@ was executed.
SWAP	(16b1 16b2 --- 16b2 16b1)	Exchanges positions of the top two items of the stack.

232. String Operations

Word	**Stack Effect**	**Description**
-TRAILING	(addr +n1 --- addr +n2)	Counts +n1 characters starting at addr and subtracts 1 from the count when a blank is encountered. Leaves on the stack the final string count, n2 and addr.
."	(---)	Displays the characters following it up to the delimiter " .
.(	(---)	Displays string following .(delimited by) .
COUNT	(addr1 --- addr2 +n)	Leaves the address, addr2 and the character count +n of text beginning at addr1.

PCOUNT	(addr1 --- addr2 +n)	Leaves the address, addr2 and the character count +n of text beginning at addr1 in Program memory.

233. Terminal I/O

Word	Stack Effect	Description
?KEY	(--- flag)	True if any key is depressed.
?TERMINAL	(--- flag)	True if any key is depressed. Same as ?KEY.
CR	(---)	Generates a carriage return and line feed.
EMIT	(16b ---)	Displays the ASCII equivalent of 16b onto the screen.
EXPECT	(addr +n ---)	Stores up to +n characters into memory beginning at addr.
KEY	(--- 16b)	Pauses to wait for a key to be pressed and then places the ASCII value of the key (n) on the stack.
PTYPE	(addr +n ---)	Displays a string of +n characters from Program memory, starting with the character at addr.
SPACE	(---)	Sends a space (blank) to the current output device.
SPACES	(+n ---)	Sends +n spaces (blanks) to the current output device.
TYPE	(addr +n ---)	Displays a string of +n characters starting with the character at addr.

234. Numeric Output

Word	Stack Effect	Description
#	(+d1 --- +d2)	+d1 is divided by BASE and the quotient is placed onto the stack. The remainder is converted to an ASCII character and appended to the output string toward lower memory addresses.
#>	(32b --- addr +n)	Terminates formatted (or pictured) output string (ready for TYPE).
#S	(+d --- 0 0)	Converts all digits of an entire number into string.
(E.)	(F:r --)(-- addr +n)	Convert the top number on the floating-point stack to its character string representation using the scientific notation. Addr is the address of the location where the character string representation of r is stored, and +n is the number of bytes.
(F.)	(F:r --)(-- addr +n)	Convert the top number on the floating-point stack to its character string representation using the fixed-point notation. Addr is the address of the location where the character string representation of r is stored, and +n is the number of bytes.
.	(n ---)	Removes n from the top of stack and displays it.
.R	(n +n ---)	Displays the value n right justified in a field +n

		characters wide according to the value of BASE.
<#	(---)	Starts a formatted (pictured) numeric output. Terminated by #> .
?	(addr ---)	Displays the contents of addr.
BASE	(--- addr)	Leaves the address of the user variable containing the numeric numeric conversion radix.
D.	(d ---)	Displays the value of d.
D.R	(d +n ---)	Displays the value of d right justified in a field +n characters wide.
DECIMAL	(---)	Sets the input-output numeric conversion base to ten.
E.	(--)(F:r --)	Convert the top number on the floating-point stack to its character string representation using the scientific notation.
F.	(F:r --)(--)	Print the top number on the floating-point stack on the screen using fixed-point notation.
F?	(addr --)	Display the floating-point contents stored at addr.
HEX	(---)	Sets the numeric input-output conversion base to sixteen.
HOLD	(char ---)	Inserts character into a pictured numeric out- put string.
PLACES	(n ---)	Set the number of decimal places (digits to the right of the radix point) displayed by E. and F.
SIGN	(n ---)	Appends an ASCII "; - "; (minus sign) to the start of a pictured numeric output string if n is negative.
U.	(u ---)	Displays the unsigned value of u followed by a space.
U.R	(u +n ---)	Displays the value of u right justified in a field +n characters wide according to the value of BASE.

235. Numeric Input

Word	Stack Effect	Description
CONVERT	(+d1 addr1 --- +d2 addr2)	Converts an input string into a number.
FNUMBER	(+d1 addr1 -- +d2 addr2)	Converts an input string into a number.
NUMBER	(addr --- d)	Converts the counted string at addr to d according to the value of BASE .

236. Memory Operations

Word	Stack Effect	Description
!	(16b addr ---)	Stores 16b at addr.
2!	(32b addr ---)	Stores 32b at addr.

2@	(addr --- 32b)	Returns 32b from addr.
@	(addr --- 16b)	Replaces addr with its 16b contents on top of the stack.
@!	(16b addr ---)	Stores 16 at address pointed to by addr.
@@	(addr --- 16b)	Replaces addr with 16b, 16b is contents of address pointed to by addr.
BLANK	(addr u ---)	Sets u bytes of memory beginning at addr to the ASCII code for space (decimal 32).
C!	(c addr ---)	Stores the character c into addr.
C@	(addr --- c)	Fetches the character c contents from addr.
CMOVE	(addr1 addr2 u ---)	Moves towards high memory the u bytes at addresses addr1 and addr2.
CMOVE>	(addr1 addr2 u ---)	Moves u bytes beginning at addr1 to addr2.
D!	(32b addr ---)	Stores 32b at addr. Same as 2!
D@	(addr --- 32b)	Returns 32b from addr. Same as 2@
EE!	(16b addr ---)	Stores 16b into addr in EEPROM.
EEC!	(16b addr ---)	Stores the least significant byte of 16b into addr in EEPROM.
EEMOVE	(addr1 addr2 u ---)	Moves towards high memory the u bytes at addresses addr1 and addr2. addr2 should be in EEPROM.
EEERASE	(addr ---)	Erase one page of Data Flash memory at addr.
ERASE	(addr u ---)	Sets u bytes of memory to zero, beginning at addr.
EXCHANGE	(w1 addr --- w2)	Fetches contents w2 from addr, then stores w1 at addr. (Exchanges w1 for w2 at addr.)
FILL	(addr u c ---)	Fills u bytes, beginning at addr, with byte pattern c.
P!	(16b addr ---)	Stores 16b into Program memory at at addr.
P@	(addr --- 16b)	Fetches the 16b contents from Program memory at addr.
PC!	(c addr ---)	Stores the character c into Program memory at addr.
PC@	(addr --- c)	Fetches the character c contents from Program memory at addr.
PF!	(16b addr ---)	Stores 16b into addr in Program Flash ROM.
PFERASE	(addr ---)	Erase one page of Program Flash memory at addr.
PFMOVE	(addr1 addr2 u ---)	Moves the u locations from Program RAM at addr1, to Program Flash at addr2.
TOGGLE	(addr b --)	Toggles setting of bits with mask b at addr.

237. Memory Allocation

Word	Stack Effect	Description
,	(16b ---)	Stores 16b into a word at the next available dictionary location.

?AVAIL	(---)	Prints an error message if insufficient RAM or Flash memory space is available.
ALLOT	(w ---)	Reserves w bytes of dictionary space.
AVAIL	(--- n)	Returns number of locations remaining in Data RAM memory.
C,	(c ---)	Stores the character c into a byte at the next available dictionary location.
CELL+	(addr1 --- addr2)	Add the size of one cell to addr1, giving addr2.
EEAVAIL	(--- n)	Returns number of locations remaining in EEPROM (Data Flash) memory.
EXRAM	(---)	Enable external RAM. (for future use)
FLOAT+	(addr1 --- addr2)	Add the size of one floating-point number to addr1, giving addr2.
FLOATS	(n1 --- n2)	Returns the number of memory locations n2 used by n1 floating-point numbers.
HERE	(--- addr)	Leaves the address of the next available dictionary location.
P,	(w ---)	Stores 16b into a word at the next available location in Program memory.
PALLOT	(n ---)	Reserves n bytes of dictionary space in Program memory.
PAVAIL	(--- n)	Returns number of locations remaining in Program RAM memory.
PC,	(c ---)	Stores the character c into a byte at the next available location in Program memory.
PF,	(n ---)	Stores 16b into a word at the next available location in Program Flash ROM.
PFAVAIL	(--- n)	Returns number of locations remaining in Program Flash memory.
PHERE	(--- addr)	Leaves the address of the next available dictionary location in Program memory.

238. Program Control

Word	**Stack Effect**	**Description**
+LOOP	(n ---) *(C: sys ---)*	Increments the DO LOOP index by n.
AGAIN	(---) *(C: sys ---)*	Affect an unconditional jump back to the start of a BEGIN-AGAIN loop.
BEGIN	(---) *(C: --- sys)*	Marks the start of a loop.
DO	(w1 w2 ---) *(C: --- sys)*	Repeats execution of words between DO LOOPs and DO +LOOPs, the number of times is specified by the limit from w2 to w1.
ELSE	(---)	Allows execution of words between IF and ELSE if

	(C: sys1 --- sys2)	the flag is true, otherwise, it forces execu- tion of words after ELSE.
END	(flag ---) *(C: sys ---)*	Performs the same function as UNTIL . See UNTIL .
EXECUTE	(addr ---)	Executes the definition found at addr.
EXIT	(---)	Causes execution to leave the current word and go back to where the word was called from.
I	(--- w)	Places the loop index onto the stack.
IF	(flag ---) *(C: --- sys)*	Allows a program to branch on condition.
J	(--- w)	Returns the index of the next outer loop.
K	(--- w)	Returns the index of the second outer loop in nested do loops.
LEAVE	(---)	Forces termination of a DO LOOP.
LOOP	(---) *(C: sys ---)*	Defines the end point of a do-loop.
REPEAT	(---) *(C: sys ---)*	Terminates a BEGIN...WHILE...REPEAT loop.
THEN	(---) *(C: sys ---)*	Marks the end of a conditional branch or marks where execution will continue relative to a cor- responding IF or ELSE .
UNTIL	(flag ---) *(C: sys ---)*	Marks the end of an indefinite loop.
WHILE	(flag ---) *(C: sys1 --- sys2)*	Decides the continuation or termination of a BEGIN...WHILE...REPEAT loop.

239. Compiler

‘ <name>	(--- addr)	Returns <name>'s compilation address, addr.
(	(---)	Starts a comment input. Comment is ended by a) .
: <name>	(--- sys)	Starts the definition of a word <name>. Definition is terminated by a ; .
:CASE	(n ---) *(C: --- sys)*	Creates a dictionary entry for <name> and sets the compile mode.
;	(sys ---)	Terminates a colon-definiton.
;CODE	(---) *(C: sys1 --- sys2)*	Terminates a defining-word. May only be used in compilation mode.
AUTOSTART <name>	(addr ---)	Prepare autostart vector at addr which will cause <name> to be executed upon reset. Note: addr must be on a 1K address boundary.
CODE	(--- sys)	Creates an assembler definition.
CODE-INT	(--- sys)	Creates an assembler definition interrupt routine.
CODE-SUB	(--- sys)	Creates an assembler definition subroutine.
COMPILE	(---)	Copies the compilation address of the next non-

		immediate word following COMPILE.
CONSTANT <name>	(16b ---)	Creates a dictionary entry for <name>.
DOES>	(--- addr) *(C: ---)*	Marks the termination of the defining part of the defining word <name> and begins the definition of the run-time action for words that will later be defined by <name>.
EEWORD	(---)	Moves code of last defined word from the Program RAM memory to the Program Flash memory.
END-CODE	(sys ---)	Terminates an assembler definition.
FORGET <name>	(---)	Deletes <name> from the dictionary.
IMMEDIATE	(---)	Marks the most recently created dictionary entry as a word that will be executed immediately even if FORTH is in compile mode.
IS <name>	(16b ---)	Creates a dictionary entry <name> for the constant value 16b. Same as CONSTANT.
RECURSE	(---)	Compile the compilation address of definition currently being defined.
UNDO	(---)	Forget the latest definition regardless of smudge condition.
USER <name>	(n ---)	Create a user variable.
VARIABLE <name>	(---)	Creates a single length variable.
\	(---)	Starts a comment that continues to end-of-line.

240. Compiler Internals

Word	**Stack Effect**	**Description**
;S	(---)	Stop interpretation.
<BUILDS	(---)	Creates a new dictionary entry for <name> which is parsed from the input stream.
<MARK	(--- addr)	Leaves current dictionary location to be resolved by <RESOLVE .
<RESOLVE	(addr ---)	Compiles branch offset to location previously left by <MARK .
>BODY	(addr1 --- addr2)	Leaves on the stack the parameter field address, addr2 of a given field address, addr1.
>MARK	(--- addr)	Compiles zero in place of forward branch offset and marks it for future resolve.
>RESOLVE	(addr ---)	Corrects branch offset previously compiled by >mark to current dictionary location.
?BRANCH	(flag ---)	Compiles a conditional branch operation.
?COMP	(--)	Checks for compilation mode, gives error if not.
?CSP	(--)	Checks for stack integrity through defining process,

		gives error if not.
?ERROR	(flag n --)	If flag is true, error n is initiated.
?EXEC	(--)	Checks for interpretation mode, gives error if not.
?PAIRS	(n1 n2 --)	Checks for matched structure pairs, gives error if not.
?STACK	(---)	Checks to see if stack is within limits, gives error if not
[	(---)	Places the system into interpret state to execute non-immediate word/s during compilation.
[']	(--- addr) *(C: ---)*	Returns and compiles the code field address of a word in a colon-definition.
[COMPILE]	(---)	Causes an immediate word to be compiled.
]	(---)	Places the system into compilation state.] places a non-zero value into the user variable STATE.
ATO4	(--- n)	Returns address of subroutine call to high level word as indicated in R0 register.
BRANCH	(---)	Compiles an unconditional branch operation.
CFA	(pfa --- cfa)	Alter parameter field pointer address to code field address.
CREATE <name>	(---)	Creates a dictionary entry for <name>.
DLITERAL	(32b ---)	Compile a system dependent operation so that when later executed, 32b will be left on the stack.
FIND	(addr1 --- addr2 n)	Obtains an address of counted strings, addr1 from the stack. Searches the dictionary for the string.
FLITERAL	(F:r --)	Compile r as a floating point literal.
INTERPRET	(---)	Begins text interpretation at the character indexed by the contents of >IN relative to the block number contained in BLK, continuing until the input stream is exhausted.
LATEST	(--- nfa)	Leaves name field address (nfa) of top word in CURRENT.
LFA	(pfaptr --- lfa)	Alter parameter field pointer address to link field address.
LITERAL	(16b ---)	Compile a system dependent operation so that when later executed, 16b will be left on the stack.
NFA	(pfaptr - nfa)	Alter parameter field pointer address to name field address.
PFAPTR	(nfa --- pfaptr)	Alter name field address to parameter field pointer address.
QUERY	(---)	Stores input characters into text input buffer.
SMUDGE	(---)	Toggles visibility bit in head, enabling definitions to be found.
TASK	(---)	A dictionary marker null word.
TRAVERSE	(addr n --- addr)	Adjust addr positively or negatively until contents of addr is greater then $7F.

WORD	(char --- addr)	Generates a counted string until an ASCII code, char is encountered or the input stream is exhausted. Returns addr which is the beginning address of where the counted string are stored.

241. Error Processing

Word	**Stack Effect**	**Description**
ABORT	(---)	Clears the data stack and performs the function of QUIT .
ABORT”	(flag ---) (C: ---)	If flag is true, message that follows "; is dis- played and the ABORT function is performed. If flag is false, the flag is dropped and execu- tion continues.
COLD	(---)	Cold starts FORTH.
ERROR	(--)	Begins error processing.
MESSAGE	(n --)	Prints error message # n.
QUIT	(---)	Clears the return stack, stops compilation and returns control to current input device.

242. System Variables

Word	**Stack Effect**	**Description**
#TIB	(--- addr)	Returns the address of the user variable that holds the number of characters input.
>IN	(--- addr)	Leaves the address of the user variable >IN which contains the number of bytes from the beginning of the input stream at any particular moment during interpretation.
BLK	(--- addr)	Leaves the address of the user variable contain- ing the the number of block that is currently being interpreted.
CONTEXT	(--- addr)	Returns the address of a user variable that determines the vocabulary to be searched first in the dictionary.
CURRENT	(--- addr)	Returns the address of the user variable specifying the vocabulary into which new word definitions will be entered.
DP	(--- addr)	Put Dictionary Pointer address on stack.
DPL	(--- addr)	Returns the address of the user variable con- taining the number of places after the frac- tional point for input conversion.
EDELAY	(--- addr)	Put EEPROM programming delay variable onto the stack.
EDP	(--- addr)	Put EEPROM memory pointer onto the stack.

FENCE	(--- addr)	System variable which specifies the highest address from which words may be compiled.
FLD	(--- addr)	Returns the address of the user variable which contains the value of the field length reserved for a number during output conversion.
FSP	(-- addr)	User variable holds floating-point stack pointer.
FSP0	(-- addr)	User variable holds initial value of floating- point stack pointer.
PAD	(--- addr)	Puts onto stack the starting address in memory of scratchpad.
PDP	(--- addr)	System variable which holds the address of the next available Program memory location.
PFDP	(--- addr)	System variable which holds the address of the next available Program Flash memory location.
R0	(-- addr)	Returns the address of the variable containing the initial value of the bottom of the return stack.
S0	(--- addr)	Returns the address of the variable containing the initial value of the bottom of the stack.
seed	(--- addr)	Place the variable on the stack.
SPAN	(--- addr)	Returns the address of the user variable that contains the count of characters received and stored by the most recent execution of EXPECT .
STATE	(--- addr)	Returns the address of the user variable that contains a value defining the compilation state.
TIB	(--- addr)	Returns the address of the start of the text- input buffer.
UABORT	(-- addr)	User variable points to ABORT routine.
WARNING	(--)	User variable controls error handling.

243. System Constants

Word	**Stack Effect**	**Description**
B/BUF	(--- n)	Number of characters in a block storage buffer (not used).
BL	(--- 32)	Puts the ASCII code for a space (decimal 32) on the stack.
C/L	(--- n)	Maximum number of characters per line.
FALSE	(--- flag)	Returns a false flag (zero).
ISOMAX	(--- n)	Returns the current IsoMax version number.
TRUE	(--- flag)	Returns a true flag (all bits ‘1’).

244. IsoPod Control

Word	Stack Effect	Description
DINT	(---)	Disable CPU interrupts. *Warning: disables IsoMax and may disable serial I/O.*
EINT	(---)	Enable CPU interrupts.
HALFSPEEDCPU	(---)	Switch IsoPod CPU to 20 MHz clock. All timing functions (baud rate, PWM output, etc.) operate at half speed.
FULLSPEEDCPU	(---)	Switch IsoPod CPU to normal 40 MHz clock.
RESTORE-RAM	(---)	Restores system and user RAM variables from Data Flash.
SAVE-RAM	(---)	Copies system and user RAM variables to Data Flash.
SCRUB	(---)	Erases Data Flash and user's Program Flash, empties the dictionary, and restores system variables to their default values.

245. Debugging

Word	Stack Effect	Description
.S	(---)	Display stack contents without modifying the stack.
DUMP	(addr u ---)	Displays u bytes of data memory starting at addr.
F.S	(--)	Display the contents of the floating-point stack without modifying the stack.
FLASH	(---)	Launch the Flash memory programmer. (unused)
ID.	(nfa ---)	Print <name> given name field address (NFA).
PDUMP	(addr u ---)	Displays u bytes of Program memory starting at addr.
WORDS	(---)	Lists all the words in the CURRENT vocabulary.

246. Object Oriented Programming

Word	Stack Effect	Description
.CLASSES	(---)	Display all defined objects and classes. Same as WORDS.
BEGIN-CLASS <name>	(--- sys1)	Defines a class <name>, and begins the "private" definitions of the class.
END-CLASS <name>	(sys2 ---)	Ends the definition of class <name>.
NO-CONTEXT	(---)	Clears the object context, and hides all private methods.
OBJECT <name>		Defines an object <name> which is a member of the currently active class.

OBJREF	(--- addr)	System variable holding the address of the currently active object.
PUBLIC	(sys1 --- sys2)	Ends the "private" and starts the "public" definitions of the class.
SELF	(--- addr)	Returns the address of the currently active object.

247. IsoMax State Machines

Word	**Stack Effect**	**Description**
WITH-VALUE n	(--- sys)	Specifies 'n' to be used as tag value to be stored for this state.
AS-TAG	(sys ---)	Ends a tag definition for a state.
END-MACHINE-CHAIN	(sys ---)	Ends definition of a machine chain.
MACHINE-CHAIN <name>	(--- sys)	Starts definition of a machine chain <name>.
.MACHINES	(---)	Prints a list of all INSTALLed machines.
PERIOD	(n ---)	Changes the running IsoMax period to 'n' cycles.
ISOMAX-START	(---)	Initializes and starts IsoMax. Clears the machine list and starts the timer interrupt at the default rate of 50000 cycles.
NO-MACHINES	(---)	Clears the IsoMax machine list.
ALL-MACHINES	(---)	Execute, once, all machines on the IsoMax machine list.
UNINSTALL	(---)	Removes the last-added machine from the list of running IsoMax machines.
INSTALL <name>	(---)	Adds machine <name> to the list of running IsoMax machines.
MACHINE-LIST	(--- addr)	System variable pointing to the head of the IsoMax installed-machine list.
SCHEDULE-RUNS <name>	(sys ---)	Specifies that machine chain <name> is to be performed by IsoMax. This overrides the INSTALL machine list.
CYCLES	(--- sys)	Specifies period for SCHEDULE-RUNS; e.g., EVERY n CYCLES SCHEDULE-RUNS name.
EVERY	(--- sys)	Specifies period for SCHEDULE-RUNS; see CYCLES.
STOP-TIMER	(---)	Halts IsoMax by stopping the timer interrupt.
TCFAVG	(--- addr)	System variable holding the average IsoMax processing time.
TCFMIN	(--- addr)	System variable holding the minimum IsoMax processing time.
TCFMAX	(--- addr)	System variable holding the maximum IsoMax processing time.

TCFALARMVECTOR	(--- addr)	System variable holding the CFA of a word to be performed when TCFALARM is reached. Zero means "no action."
TCFALARM	(--- addr)	System variable holding an "alarm limit" for TCFOVFLO. Zero means "no alarm."
TCFOVFLO	(--- addr)	System variable holding a count of the number of times state processing overran the allotted time.
TCFTICKS	(--- addr)	System variable holding a running count of IsoMax timer interrupts.
IS-STATE?	(addr --- f)	Given state address "addr", returns true if that is the current state in the associated state machine.
SET-STATE	(addr ---)	Makes the given state "addr" the current state in its associated state machine.
IN-EE	(---)	Moves code of last defined CONDITION clause from the Program RAM memory to the Program Flash memory.
TO-HAPPEN	(addr ---)	Makes given state "addr" execute on the next iteration of the IsoMax machine. Same as NEXT-TIME.
NEXT-TIME	(addr ---)	Makes given state "addr" execute on the next iteration of the IsoMax machine.
THIS-TIME	(addr ---)	Makes given state "addr" execute on this iteration of the IsoMax machine, i.e., immediately.
THEN-STATE	(sys3 ---)	Ends the CAUSES clause.
CAUSES	(sys2 --- sys3)	Specifies actions to be taken when the CONDITION clause is satisfied.
CONDITION	(sys1 --- sys2)	Specifies the logical condition to be tested for a state transition.
IN-STATE	(--- sys1)	Specifies the state to which the following condition clause will apply.
ON-MACHINE <name>	(---)	Specifies the machine to which new states and condition clauses will be added.
APPEND-STATE <name>	(---)	Adds a new state "name" to the currently selected machine.
MACHINE <name>	(---)	Defines a new state machine "name".
CURSTATE	(--- addr)	System variable used by the IsoMax compiler.
ALLOC	(n --- addr)	Allocate "n" locations of state data and return its address "addr".
RAM	(--- addr)	System variable which holds an optional address for IsoMax state data allocation. If zero, IsoMax state data will use the dictionary for state data.

248. I/O Trinaries

Word	Stack Effect	Description
AND-MASK n	(--- sys)	Specifies ‘n’ to be used as the AND mask for output.
AT-ADDR addr	(--- sys)	Specifies the address ‘addr’ to be used for input or output.
CLR-MASK n	(--- sys)	Specifies ‘n’ to be used as the Clear mask for output.
DATA-MASK n	(--- sys)	Specifies ‘n’ to be used as the Data mask for input.
DEFINE <name>	(--- sys1)	Begin the definition of an I/O or procedural trinary.
END-PROC	(sys2 ---)	Ends a PROC definition.
FOR-INPUT	(sys ---)	Ends an input trinary definition.
FOR-OUTPUT	(sys ---)	Ends an output trinary definition.
PROC	(sys1 --- sys2)	Defines an I/O trinary using procedural code.
SET-MASK n	(--- sys)	Specifies ‘n’ to be used as the Set mask for output.
TEST-MASK n	(--- sys)	Specifies ‘n’ to be used as the Test mask for input.
XOR-MASK n	(--- sys)	Specifies ‘n’ to be used as the XOR mask for output.

249. Loop Indexes

Word	Stack Effect	Description
LOOPINDEX <name>	(---)	Define a loop-index variable <name>. <name> will then be used to select the variable for one of the following index operations.
START	(n ---)	Set the starting value of the given loop-index variable.
END	(n ---)	Set the ending value of the given loop-index variable.
STEP	(n ---)	Set the increment to be used for the given loop-index variable.
RESET	(---)	Reset the given loop-index variable to its starting value.
COUNT	(--- flag)	Increment the loop-index variable by its STEP value. If it passes the END value, reset the variable and return a true flag. Otherwise return a false flag.
VALUE	(--- n)	Return the current value of the given loop-index variable.
LOOPINDEXES	(---)	Select LOOPINDEXES methods for compilation.

250. Bit I/O

Word	Stack Effect	Description
PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7 PD0 PD1 PD2 PD3 PD4 PD5 PB0 PB1 PB2 PB3 PB4 PB5 PB6 PB7 PA0 PA1 PA2 PA3 PA4 PA5 PA6 PA7 GRNLED YELLED REDLED	(---)	Select the given pin or LED for the following I/O operation.
OFF	(---)	Make the given pin an output and turn it off.
ON	(---)	Make the given pin an output and turn it on.
TOGGLE	(---)	Make the given pin an output and invert its state.
SET	(flag ---)	Make the given pin an output and set it on or off as determined by flag.
GETBIT	(--- 16b)	Make the given pin an input and return its bit value.
ON?	(--- flag)	Make the given pin an input and return true if it is on.
OFF?	(--- flag)	Make the given pin an input and return true if it is off.
?ON	(--- flag)	Return true if the pin is on; do not change its direction (works with input or output pins).
?OFF	(--- flag)	Return true if the pin is off; do not change its direction (works with input or output pins).
IS-OUTPUT	(---)	Make the given pin an output.
IS-INPUT	(---)	Make the given pin an input. (Hi-Z)
I/O <name>	(16b addr ---)	Define a GPIO pin <name> using bit mask 16b at addr.
GPIO	(---)	Select GPIO methods for compilation.

251. Byte I/O

Word	Stack Effect	Description
PORTB PORTA	(---)	Select the given port for the following I/O operation.
GETBYTE	(--- 8b)	Make the given port an input and return its 8-bit contents as 8b.
PUTBYTE	(8b ---)	Make the given port an output and write the value 8b to the port.
IS-OUTPUT	(---)	Make the given port an output.
IS-INPUT	(---)	Make the given port an input. (Hi-Z)
I/O <name>	(addr ---)	Define a GPIO port <name> at addr.

BYTEIO	(---)	Select BYTEIO methods for compilation.

252. Serial Communications Interface

Word	Stack Effect	Description
SCI1 SCI0	(---)	Select the given port for the following I/O operation.
BAUD	(u ---)	Set the serial port to “u” baud. If HALFSPEEDCPU is selected, the baud rate will be u/2.
RX?	(--- u)	Return nonzero if a character is waiting in the receiver. If buffered, return the number of characters waiting.
RX	(--- char)	Get a received character. If no character available, this will wait.
TX?	(--- u)	Return nonzero if the transmitter can accept a character. If buffered, return the number of characters the buffer can accept.
TX	(char ---)	Send a character.
RXBUFFER	(addr u ---)	Specify a buffer at addr with length u is to be used for receiving. u must be at least 5. If u=0, disables receive buffering.
TXBUFFER	(addr u ---)	Specify a buffer at addr with length u is to be used for transmitting. u must be at least 5. If u=0, disables transmit buffering.
SCIS	(---)	Select SCIS methods for compilation.

253. Serial Peripheral Interface

Word	Stack Effect	Description
SPI0	(---)	Select the given port for the following I/O operation.
MBAUD	(n ---)	Set the SPI port to n Mbaud. n must be 1, 2, 5, or 20, corresponding to actual rates of 1.25, 2.5, 5, or 20 Mbaud. All other values of n will be ignored and will leave the baud rate unchanged.
LEADING-EDGE	(---)	Receive data is captured by master & slave on the first (leading) edge of the clock pulse. (CPHA=0)
TRAILING-EDGE	(---)	Receive data is captured by master & slave on the second (trailing) edge of the clock pulse. (CPHA=1)
ACTIVE-HIGH	(---)	Leading and Trailing edge refer to an active-high pulse. (CPOL=0).
ACTIVE-LOW	(---)	Leading and Trailing edge refer to an active-low pulse. (CPOL=1).
LSB-FIRST	(---)	Cause data to be sent and received LSB first.
MSB-FIRST	(---)	Cause data to be sent and received MSB first.

BITS	(n ---)	Specify the word length to be transmitted/received. n may be 2 to 16.
SLAVE	(---)	Enable the port as an SPI slave.
MASTER	(---)	Enable the port as an SPI master.
RX-SPI?	(--- u)	Return nonzero if a word is waiting in the receiver. If buffered, return the number of words waiting.
RX-SPI	(--- 16b)	Get a received word. If no word is available in the receive buffer, this will wait. 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.
TX-SPI?	(--- u)	Return nonzero if the transmitter can accept a word. If buffered, return the number of words the buffer can accept.
TX-SPI	(16b ---)	Send a word on the SPI port. In `MASTER` mode, this will output 2 to 16 bits on the MOSI, generate 2 to 16 clocks on the SCLK pin, and simultaneously input 2 to 16 bits on the MISO pin.
RXBUFFER	(addr u ---)	Specify a buffer at addr with length u is to be used for receiving. u must be at least 5. If u=0, disables receive buffering.
TXBUFFER	(addr u ---)	Specify a buffer at addr with length u is to be used for transmitting. u must be at least 5. If u=0, disables transmit buffering.
SPI	(---)	Select SPI methods for compilation.

254. Timers

Word	Stack Effect	Description
TD2 TD1 TD0 TC3 TC2 TC1 TC0 TB3 TB2 TB1 TB0 TA3 TA2 TA1 TA0	(---)	Select the given timer for the following I/O operation.
ACTIVE-HIGH	(---)	Change output & input to normal polarity, 1=on. For output, `PWM-OUT` will control the *high* pulse width. For input, `CHK-PWM-IN` will measure the width of the *high* pulse. The reset default is `ACTIVE-HIGH`.
ACTIVE-LOW	(---)	Change output & input to inverse polarity, 0=on. For output, `PWM-OUT` will control the *low* pulse width. For input, `CHK-PWM-IN` will measure the width of the *low* pulse.
ON	(---)	Make the given pin a digital output and turn it on.

OFF	(---)	Make the given pin a digital output and turn it off.
TOGGLE	(---)	Make the given pin a digital output and invert its state.
SET	(flag ---)	Make the given pin a digital output and set it on or off as determined by flag.
ON?	(--- flag)	Make the given pin a digital input and return true if it is on.
OFF?	(--- flag)	Make the given pin a digital input and return true if it is on.
GETBIT	(--- 16b)	Make the given pin a digital input and return its bit value.
?ON	(--- flag)	Return true if the timer input pin is on; do not change its mode.
?OFF	(--- flag)	Return true if the timer input pin is off; do not change its mode.
SET-PWM-IN	(---)	Start time measurement of an input pulse. The duration of the next high pulse will be measured (or low pulse if ACTIVE-LOW).
CHK-PWM-IN	(--- u)	Returns the measured duration of the pulse, in cycles of a 2.5 MHz clock, or zero if not yet detected. Only the first non-zero result is valid; successive checks will give indeterminate results.
PWM-PERIOD	(u ---)	Set PWM period to u cycles of a 2.5 MHz clock. u may be 100-FFFF hex.
PWM-OUT	(u ---)	Outputs a PWM signal with a given duty cycle u, 0-FFFF hex, where FFFF is 100%. `PWM-PERIOD` must be specified before using `PWM-OUT`.
TIMER <name>	(addr ---)	Define a timer <name> at addr.
TIMERS	(---)	Select TIMERS methods for compilation.

255. PWM Output Pins

Word	**Stack Effect**	**Description**
PWMB5 PWMB4 PWMB3 PWMB2 PWMB1 PWMB0 PWMA5 PWMA4 PWMA3 PWMA2 PWMA1 PWMA0	(---)	Select the given pin for the following I/O operation.
PWM-PERIOD	(+n ---)	Set PWM period to +n cycles of a 2.5 MHz clock. n may be 100-7FFF hex. This will affect all PWM outputs in the group (A or B).
PWM-OUT	(u ---)	Outputs a PWM signal with a given duty cycle u, 0-FFFF hex, where FFFF is 100%. `PWM-PERIOD` must be specified before using `PWM-OUT`.

ON	(---)	Make the given pin a digital output and turn it on.
OFF	(---)	Make the given pin a digital output and turn it off.
TOGGLE	(---)	Make the given pin a digital output and invert its state.
SET	(flag ---)	Make the given pin a digital output and set it on or off as determined by flag.
?OFF	(--- flag)	Return true if the pin is on.
?ON	(--- flag)	Return true if the pin is off.
PWM <name>	(16b1 16b2 n addr ---)	Define a PWM output pin <name> using configuration pattern 16b1, bit pattern 16b2, and channel n, at addr.
PWMOUT	(---)	Select PWMOUT methods for compilation.

256. PWM Input Pins

Word	Stack Effect	Description
ISB2 ISB1 ISB0 FAULTB3 FAULTB2 FAULTB1 FAULTB0 ISA2 ISA1 ISA0 FAULTA3 FAULTA2 FAULTA1 FAULTA0		Select the given pin for the following I/O operation.
ON?	(--- flag)	Return true if the given pin is on.
OFF?	(--- flag)	Return true if the given pin is off.
?ON	(--- flag)	Return true if the given pin is on. Same as ON?
?OFF	(--- flag)	Return true if the given pin is off. Same as OFF?
GETBIT	(--- 8b)	Return the bit value of the given pin.
PWM <name>	(16b addr ---)	Define a PWM input pin <name> using bit mask 16b at addr.
PWMIN	(---)	Select PWMIN methods for compilation.

257. Analog-to-Digital Converter

Word	Stack Effect	Description
ADC7 ADC6 ADC5 ADC4 ADC3 ADC2 ADC1 ADC0	(---)	Select the given pin for the following I/O operation.
ANALOGIN	(--- +n)	Perform an A/D conversion on the selected pin, and return the result +n. 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.
ADC-INPUT <name>	(n addr ---)	Define an analog input pin <name> for channel n at addr.
ADCS	(---)	Select ADCS methods for compilation.

258. GLOSSARY OF TERMS

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