IsoMax™ for ARM • User's Manual



Warranty

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

Copyrights and Trademarks

Copyright ' 2002 by New Micros, Inc. All rights reserved. IsoMax“ for ARM, 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.

Disclaimer of Liability

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

Internet Access

Web site: http://www.newmicros.com
This manual: http://www.newmicros.com/store/product_manual/IsoMax for ARM.zip
Email technical questions: nmitech@newmicros.com
Email sales questions: nmisales@newmicros.com
Also see Manufacturer information near the end of this manual.

Internet IsoMax“ Discussion List

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

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

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

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

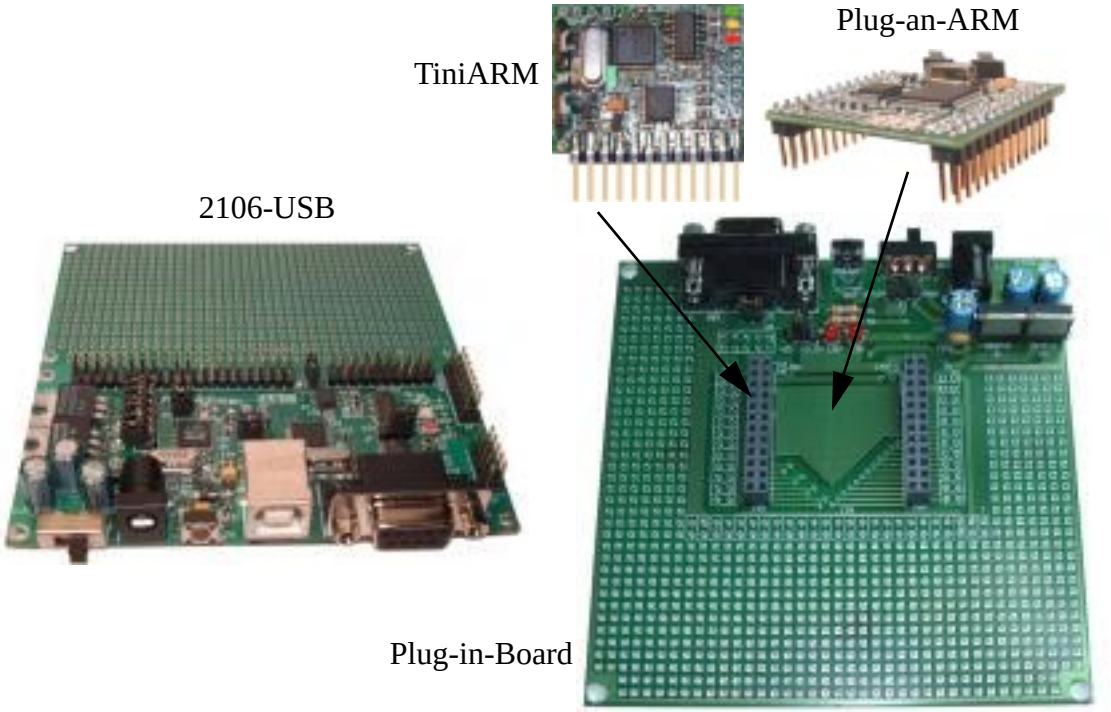
Table of Contents

1. GETTING STARTED 5
2. INTRODUCTION 9
3. QUIK TOUR 11
4. PROGRAMMING 16
   - 4.1 THREE MACHINES 18
     - 4.1.1 REDTRIGGER 18
   - 4.2 ANDGATE1 24
   - 4.3 BOUNCELESS 27
   - 4.4 SYNTAX AND FORMATTING 30
   - 4.5 MULTIPLE STATES/MULTIPLE TRANSITIONS 31
   - 4.6 ANDGATE2 31
   - 4.7 ANDOUT 33
   - 4.8 ANDGATE3 34
   - 4.9 INTER-MACHINE COMMUNICATIONS 36
   - 4.10 STATE MEMORY 36
   - 4.11 BOUNCELESS+ 37
   - 4.12 DELAYS 39
   - 4.13 BLINKGRN 40
   - 4.14 SPEED 42
   - 4.15 ZIPGRN 42
   - 4.16 REDYEL 44
   - 4.17 TRINARIES 46
   - 4.18 FLASH AND AUTOSTARTING 48
     - 4.18.1 Autovectors: 49
     - 4.18.2 Dimensions 50
     - 4.18.3 Updates 50
5. SOFTWARE 51
   - 5.1 WORD SYNTAX 52
   - 5.2 Queues 54
     - 5.2.1 Queues for supporting data and control flows 54
     - 5.2.2 Queues as a component of Data Flow 55
     - 5.2.3 Word Definitions 55
   - 5.3 Timeouts 56
     - 5.3.1 Word Definitions 56
   - 5.4 Other Definitions 56
   - 5.5 ASMs and SSMs 57

5.6 More Tricks 60
5.7 From Loops to States 60
5.8 Tools 61
5.8.1 Autoboot 61
5.8.2 Text: 62
5.8.3 Debugging: 62
5.8.4 Asynchronous State Machines: 62
5.8.5 Status: 62
5.8.6 Measurement: 62
5.9 ASM Example: Stoplight 62
5.10 Word Listing (LPC2106) 64
6. REGISTERS 69
6.1 A/D Converter 69
6.2 Can Bus 69
6.3 External Interrupts 69
6.4 I2C Interface 69
6.5 General Purpose Input/Output 70
6.6 Memory Accelerator Module 70
6.7 Power Control 70
6.8 Pin Connect Block 70
6.9 Phase Locked Loop 70
6.10 Pulse Width Modulation 70
6.11 Real Time Clock 70
6.12 Serial Peripheral Interface 71
6.13 Timers 71
6.14 Universal Asynchronous Receivers Transmitters 71
6.15 Vectored Interrupt Controller 71
6.16 VPB Divider 72
6.17 Watchdog 72
7. MEMORY MAP 73
7.1 Flash 73
8. Hands On 75
8.1 Setup 75
8.1.1 Small Flash Writes 75
9. Examples 76
9.1 Using the ADC to Measure Battery Life 76
9.1.1 Code 76
9.1.2 Results 77
9.2 Using PWM to Generate High Voltage 82
9.2.1 Code 83
9.2.2 Discussion 84
9.3 Software UART for GPS 84
9.4 Change serial port speed at Bootup 87

1. GETTING STARTED

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

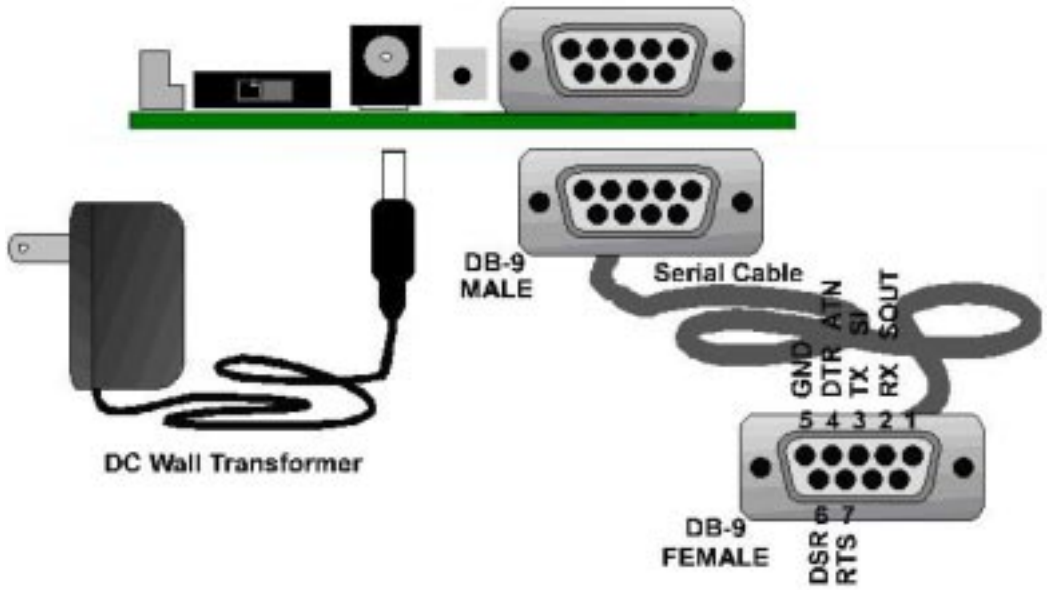


If you are new to IsoMax™ for ARM, we know you will be in a hurry to see it working.

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 "ARM" work for us!

We'll need PC running a terminal program. Then we'll need a serial cable to connect from the PC to IsoMax™ for ARM (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 IsoMax™ for ARM interactively.

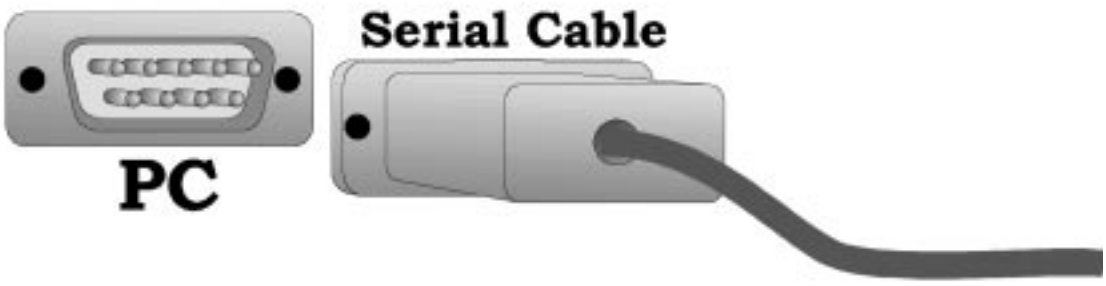


These connections are made on the serial DB9 on the ARM, and the barrel connector for accepting the 6 VDC power.

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

Start with the PC: Install and run the MaxTerm program for DOS, or NMITerm for Windows. Set the terminal program for communications channel (COM1, COM2, etc.) you wish to use, and set communications settings to (115200 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 or NMITerm or 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.



All three LED's should come on at power up if the power switch is on and power cable connected or at reset but when running only the green led is lit. If the LED's do not light, unplug the power or switch off the board 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 board. Go back through the instructions again. Check the power connections, particularly for polarity. (The outer ring of the barrel connector should be negative or ground, the inner connection should be positive 6 volts.) If the LED's come on but there is no communication, check the terminal program. Check the serial connections, to make sure everything is plugged in, and that you are using an ordinary serial cable. A null modem cable will not work. 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 "IsoMaxTM Vx.x". Odds are you're there. Congratulations! Now let's do something interactive with the board.

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 ARM.

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

```
REDLED ON
```

To turn it back off type:

```
REDLED OFF
```

Now you should have a good feeling because you can tell your ARM is working.
It's time for an overview of what your IsoMax™ for ARM has for features.
First though, a few comments on IsoMax™ revision level. The first port of IsoMax™ 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. IsoMax™

was ported from the IsoPod to the HC12, then the HCS12 and now onto the 32 bit environment of the ARM. IsoMax™ runs on both the LPC2129 on the Plug-an-ARM board, and on the LPC2106 on the USB and TiniARM boards. The two ARMs offer different hardware configurations for different environments. The ARM is a different platform than the IsoPod and the different features are discussed in detail later. As IsoMax™ grows, new features and new platforms will be added.

2. INTRODUCTION

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

What is neat about the IsoMax™ for ARM? Several things. First it is a very good micro controller. The IsoMax™ for ARM 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?

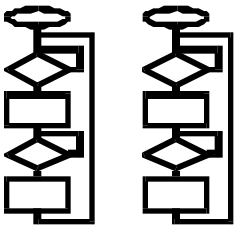
The answer is very important. The neatest thing about the IsoMax™ for ARM is software giving Virtually Parallel Machine Architecture!

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. IsoMax facilitates fine grained parallelism.



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

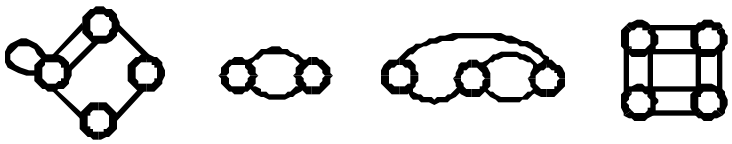
Getting two things done at the same time gets tricky. Add a few more things concurrently competing for processor attention, and most projects start running into serious trouble. Much beyond that, and only the best programmers can weave a program together running many tasks in one application.



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

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

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



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

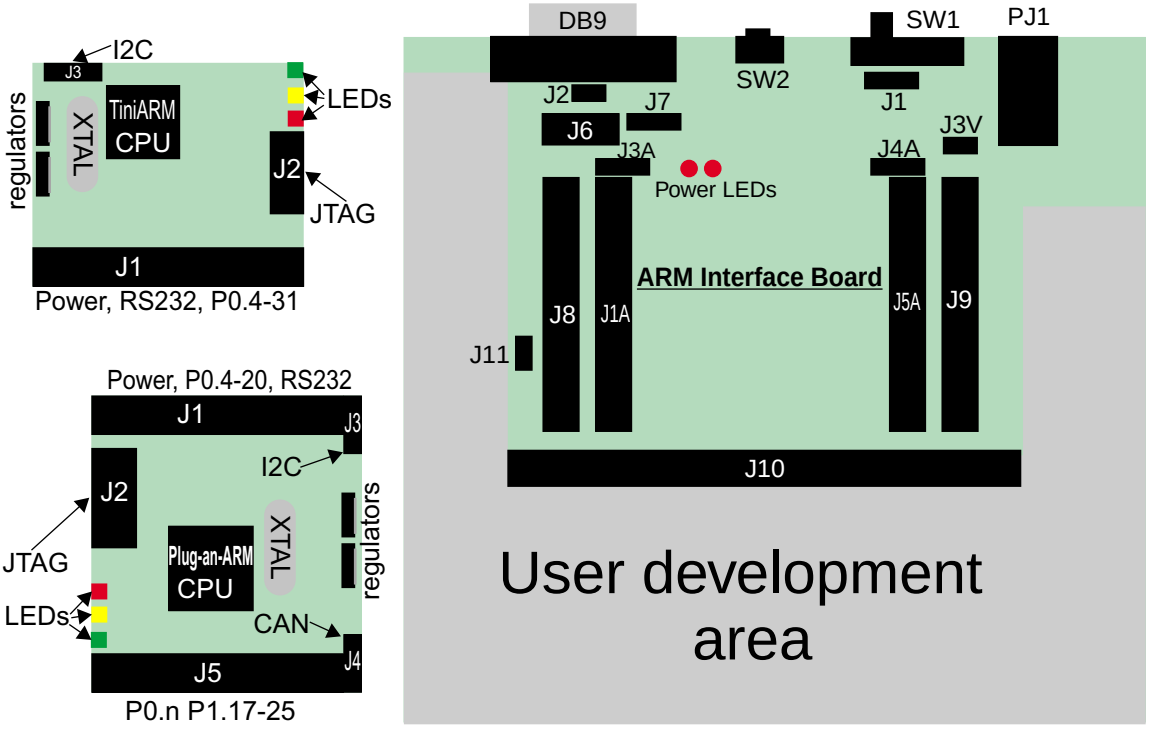
Programming means create a new processor task fashioned as a machine, and debug it interactively in the foreground. When satisfied with performance, you install the new machine in a chain of machines. The machine chain becomes a background feature of IsoMax™ until you remove it or replace it.

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

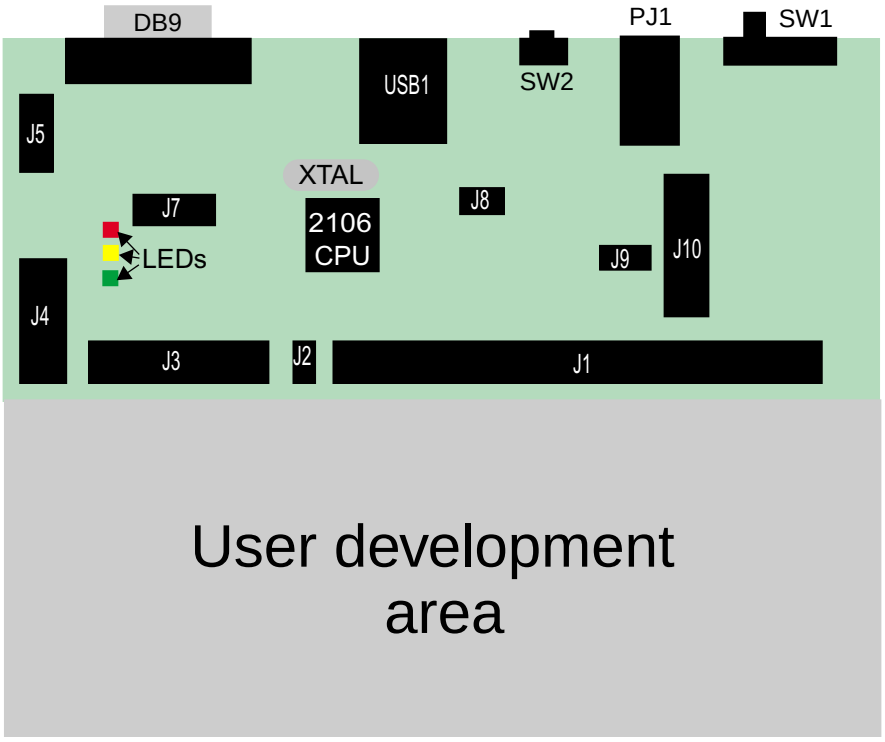
IsoMax™ brings an amazing amount power to a very small space, at a very reasonable cost. You'll undoubtedly want to have a few ARMs on hand for your future projects.

3. QUIK TOUR

Start by comparing your board to the diagrams below. Most of the important features on the board are labeled. There are two compact ARM boards which plug into an ARM Interface Board for development (Plug-an-ARM and TiniARM):



And there is the larger all in one ARM board with a USB port (LPC2106-USB):



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

Table 1: TiniARM, Plug-an-ARM and Interface Board header functions.

Board	Connector	Functions
TiniARM	J1	Power, Reset, Serial I/O, general purpose I/O for port 0
	J2	JTAG connector, or shared GPIOs
	J3	I^2C connector, or shared GPIOs
Plug-an-ARM	J1	Power, Reset, Serial I/O, general purpose I/O for port 0
	J2	JTAG connector, or shared GPIO
	J3	I2C connector, or shared GPIO
	J4	CAN BUS network port
	J5	General I/O for ports 0 and 1
ARM Interface Board	J1	Default pin 2 & 3 select on board or on plug-in board regulators. Pin 1 & 2 select the off board regulators, Q1 & Q2 on the Interface board to power the Plug-an-ARM. This is an optional when there is no regulators on the ARM module.
	J1A	receptor for plug in board connector J1
	J2	used to connect DTR to allow remote resets
	J3A	no connections
	J3V	used when plug in board has no regulator; see J7
	J4A	CAN BUS network port
	J5A	receptor for plug in board connector J5
	J6	power, reset and UART0
	J7	used when plug in board has no regulator; see J3V
	J8	access to all pins on J1A
	J9	access to all the pins on J5A
	J10	contains all signals from J8 and J9
	J11	connect a jumper to boot into boot loader

The ARM Interface Board will accept the TiniARM in the J1A connector or a Plug-an-ARM in J1A and J5A connectors. When plugging an ARM board in, be sure to always

orientate the board so the LEDs are away from the DB9 connector, or just make sure pin 1 of each connector mates to pin 1 by checking for a square solder pad. This makes a nice platform for development and programming of ARM boards which can then be plugged in the target without the development environment.

The J8, J9 and J10 connectors are best filled with female headers. This way you can directly plug in wires and components making small circuits without any soldering or wirewrapping and even jumper to an external breadboard for rapid prototyping, just like writing interactive code. The examples section makes use of this idea. Alternatively, you can use male headers which are quite handy to wire-wrap. Having J10 as well as J9 and J8 allows more connection points for each pin.

Jumper J11 is used select the boot loader program inherint on each micro. If the jumper is present, then you need to use the LPC2000 Flash Utility to do anything with the board such as reload the IsoMax™ image, or load other images. With the jumper off (it's handy to keep it on just one post so that it's there for next time), the plugged in board will boot into IsoMax™.

There are a number of options to accommodate a wide range of user choices in plug in boards. Voltage regulation for the ARM is either done on the plug in board or on the interface board. If the plug-in boards have onboard voltage regulators, then

There are a few options in the power train from the power jack to the ARM chip so we will step through them from the power jack. Power is supplied by a plug in power adaptor with center positive and between 6 and 12 volts. Switch SW1 enables power to flow to the rest of the board or not. Jumper J1 depends on the plug-in board. If the plug-in board has voltage regulators, then J1 should be set to OFFBR (voltage regulators are OFF BoaRd with respect to the Interface Board) position routing input power to the plug in board and not to the Interface Board. In the ONBR position, power is routed to the on board regulators providing 5V and 3.3V with the two red LEDs lighting With J1 in the ONBR position, J7 choses the VDD voltage for the plug-in board. With J1 in the OFFBR position, J7 should not have any connection (just put the jumper on the middle pin only). J3V is only jumpered if the plug-in board uses the second connector and has no on board regulators. Otherwise J3V should be open.

That about sums up the Interface Board except for DB9 and SW2. DB9 is where you connect your serial cable from a COM port on your PC. SW2 is the reset button which lets you start from a known position. This is sometimes necessary when playing with the I/O components and you need to return to a known state.

The TiniARM and Plug-an-ARM differ in two main ways: the amount of I/O pins and the

on-chip resources.

Table 2: Resource comparison between TiniARM and Plug-an-ARM

resource	TiniARM	Plug-an-ARM
pins	18 I/O on J1	addition 18 I/O on J5
flash memory	128KB total[a]	256K total[a]
RAM	64KB total[a]	16KB total[a]
code read protection	none	The IPA can be put in a lesser state by writing 0x87654321 at flash address 0x1FC
uarts	two	one
CAN bus	none	one port, two controllers
A/D	none	4 10-bit inputs
External memory controller	none	yes

a. bootloader, IsoMax™ and user share this space

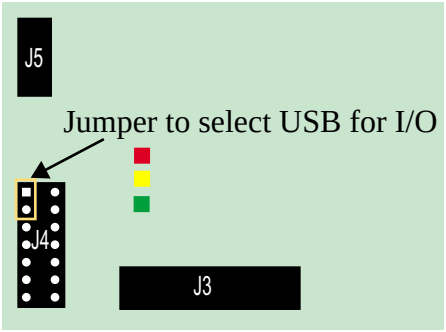
There's not too much else on the boards other than the RS232 interface chip the crystal, regulators, JTAG and I^2C ports with the Plug-an-ARM including a CAN Bus port as well. The TiniARM with regulators weighs 6 grams and the Plug-an-ARM with regulators weighs 10 grams. This puts a lot of computing power in a light weight package easily

included in robotic or other embedded applications.

Table 3: LPC2106-USB Board header functions.

Board	Connector	Functions
LPC2106 USB board	J1	all I/O ports
	J2	select DBG mode
	J3	debug port, for use with JTAG 20-Pin Connector, or shared I/O Port
	J4	debug port, for use with JTAG 14-Pin Connector or, shared I/O Port; Pin 3 & 4 select, IsoMax/FORTH boots from RS-232 (UART0) Port, DB1; J4 Pin 1 & 3 select and J7 Pin 1 & 2 select, IsoMax/FORTH (license required) boots from USB (UART1) port; J4 Pin 1 & 3 select and J7 Pin 2 & 3 select, IsoMax/FORTH (license required) boots from RS-232 (UART1) port, J5
	J5	RS-232 Connector for UART1
	J7	J7 Pin 1 & 2 select USB transmitting, or Pin 2 & 3 select RS-232 for UART1 receiving
	J8	connect a jumper to boot into boot loader
	J9	grounds PWRCTL pin on USB chip
	J10	jumpers to connect up USB to UART1

The LPC2106-USB board has a real nice feature that allows it to run from a USB cable. By jumpering the top left two pins on header J4, IsoMax™ will use the second UART for



interactive I/O. This UART is connected to a USB interface chip and creates a COM port for the PC. A USB cable connects the board to the PC and also provides power to the board so that no power supply is required. The LEDs are available for user programs.

Since there are two ports on this board, one serial port can be used for development leaving the other one available for interfacing to smart peripherals like GPS or LCDs.

4. PROGRAMMING

Under construction…

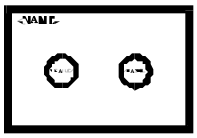
IsoMax™ is a programming language based on Finite State Machine (FSM) concepts applied to software, with a procedural language (derived from Forth) underneath it. The closest description to the FSM construction type is a “One-Hot” Mealy type of Timer Augmented Finite State Machines. More on these concepts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

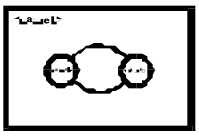
QUICK OVERVIEW

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

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

Step	Programming Action	Syntax
1	Name a state machine 	`MACHINE <name>`
2	Select this state	`ON-MACHINE <name>`

3	Name any states appended on the machine 	`APPEND-STATE <name>`
		`APPEND-STATE <name>`
		`Ö`

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.

4.1 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.

4.1.1 REDTRIGGER

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

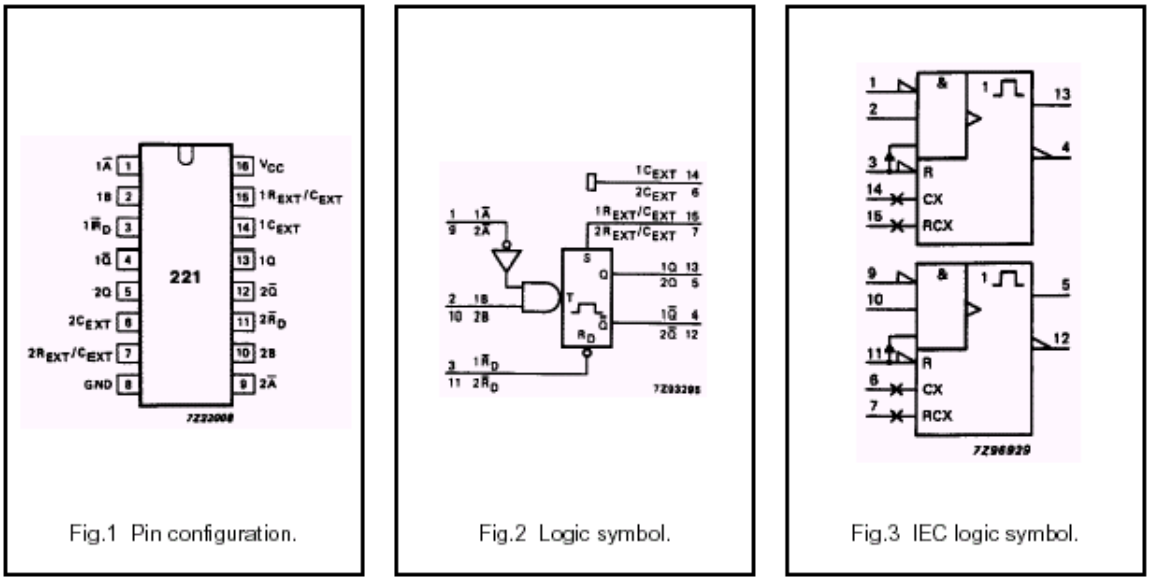
```
HEX
80000000 CONSTANT P0.31  \ pin thirty-one bit
DECIMAL

MACHINE REDTRIGGER ON-MACHINE REDTRIGGER APPEND-STATE RT
IN-STATE RT CONDITION P0.31 OFF? CAUSES REDLED ON THEN-STATE RT TO-HAPPEN

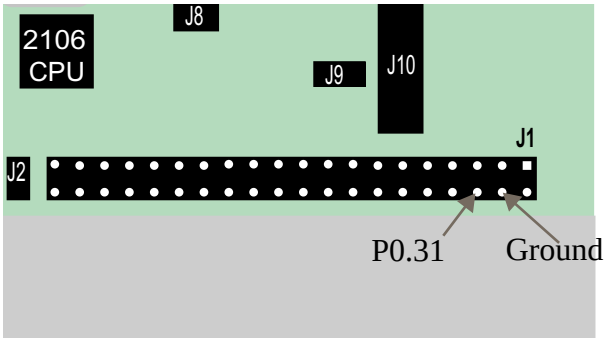
RT SET-STATE INSTALL REDTRIGGER
```

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.



If P0.31 goes low briefly, the red LED turns on and stays on even if P0.31 then changes. P0.31 normally has a pull up resistor that will keep it “on”, or “high” if nothing is attached.

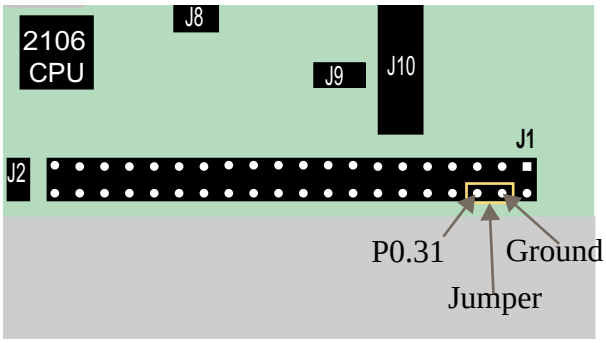


So attaching push button from P0.31 to ground, or even hooking a jumper to ground as shown, will cause P0.31 to go “off” or “low”, and the REDLED will come on.

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.)

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

```
RT SET-STATE  INSTALL REDTRIGGER
```



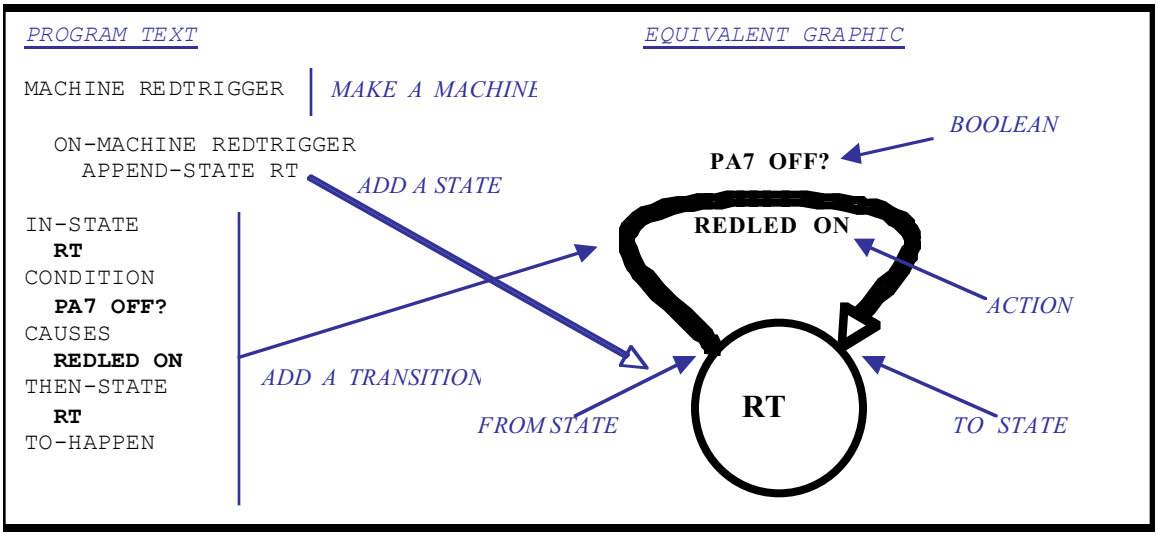
Ground P0.31 with a jumper or press the push button, and see the red LED come on. Remove the ground or release the push button. The red LED does not go back off. The program is still running, even though all visible changes end at that point. To see that, we'll need to manually reset the LED off so we can see something happen again. Enter.

```
REDLED OFF
```

If we ground P0.31 again, the red LED will come back on, so even though we are still fully interactive with IsoMax™ able to type commands like `REDLED OFF` in manually, the `REDTRIGGER` machine is running in the background.

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

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



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

```
MACHINE REDTRIGGER
```

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

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

```
ON-MACHINE REDTRIGGER
```

(Right now, we only have one machine installed. We could have skipped this second line. Since there could be several machines already in IsoMax™ 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 `P0.31 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 P0.31 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
  P0.31 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.)

4.2 ANDGATE1

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

```
HEX
40000000 CONSTANT P0.30  \ pin thirty

DECIMAL

MACHINE ANDGATE1 ON-MACHINE ANDGATE1 APPEND-STATE X
IN-STATE X CONDITION YELLED OFF P0.31 ON? P0.30 ON? AND CAUSES
   YELLED ON THEN-STATE X TO-HAPPEN

X SET-STATE  INSTALL ANDGATE1
```

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

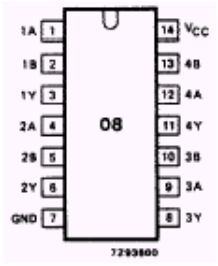


Fig.1 Pin configuration.

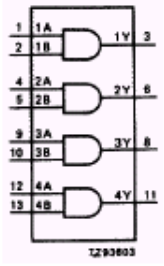


Fig.2 Logic symbol.

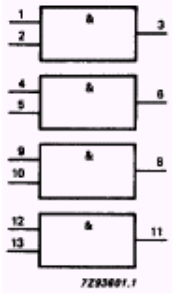


Fig.3 IEC logic symbol.

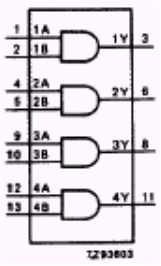


Fig.4 Functional diagram.

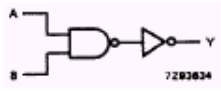


Fig.5 HC logic diagram (one gate).

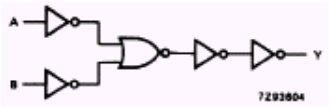


Fig.6 HCT logic diagram (one gate).

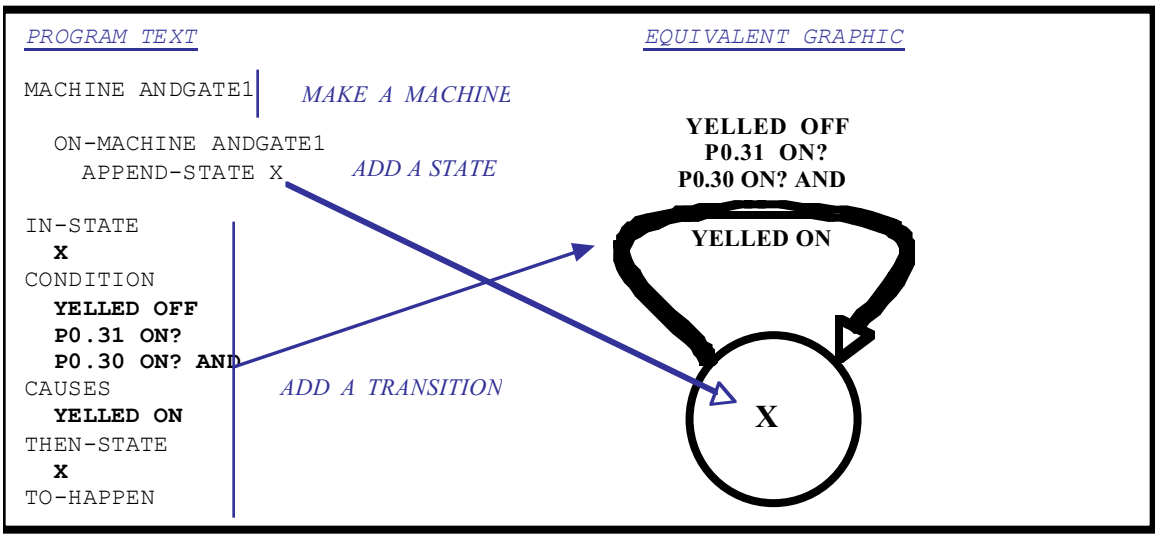
FUNCTION TABLE

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 P0.31 and P0.30 must be on, or high, to allow the yellow LED to remain on (most of the time). So by attaching push buttons to P0.31 and P0.30 simulating micro switches this little program could be used like an interlock system detecting “cover closed”.



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

Notice both machines share an input, but are using the opposite sense on that input. ANDGATE1 looks for P0.31 to be ON, or HIGH. The internal pull up will normally make P0.31 high, as long as it is programmed for a pull up and nothing external pulls it down.

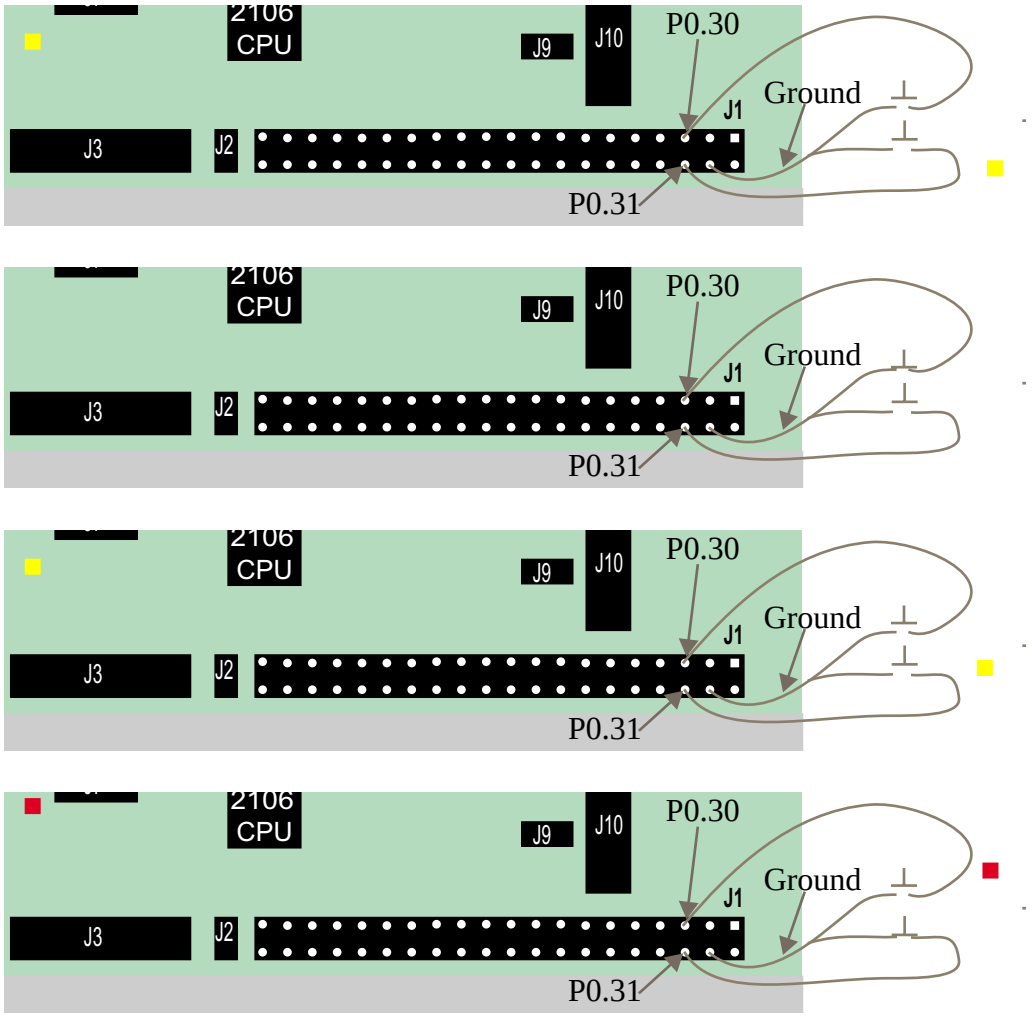
Grounding P0.31 enables REDTRIGGER's condition, and inhibits ANDGATE1's condition. Yet the two machines coexist peacefully on the same processor, even sharing the same inputs in different ways.

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 P0.31 now causes the same result for REDTRIGGER, the red LED goes on, but the opposite effect for the yellow LED, which goes off while P0.31 is grounded. Releasing P0.31 turns the yellow LED back on, but the red LED remains on.

Again, initialize REDTRIGGER by turning REDLED OFF. Now ground P0.30. This has no effect on the red LED, but turns off the yellow LED while grounded. Grounding both

P0.31 and P0.30 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 P0.31 input.

4.3 BOUNCELESS

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

```
HEX
20000000 CONSTANT P0.29  \ pin thirty

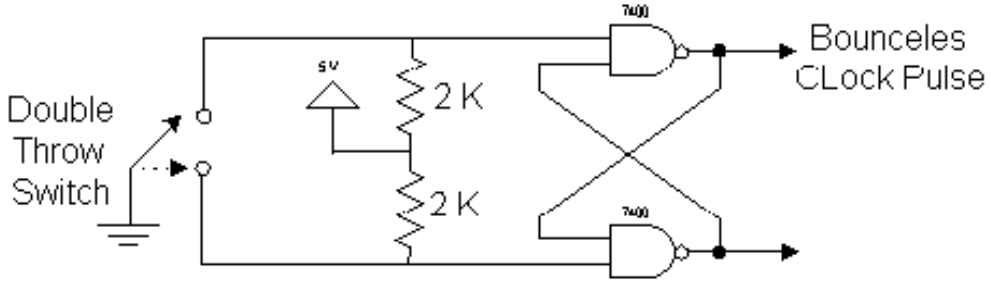
MACHINE BOUNCELESS ON-MACHINE BOUNCELESS APPEND-STATE Y
```

```
IN-STATE Y CONDITION P0.31 OFF? CAUSES GRNLED OFF THEN-STATE Y TO-HAPPEN
IN-STATE Y CONDITION P0.29 OFF? CAUSES GRNLED ON THEN-STATE Y TO-HAPPEN

Y SET-STATE  INSTALL BOUNCELESS
```

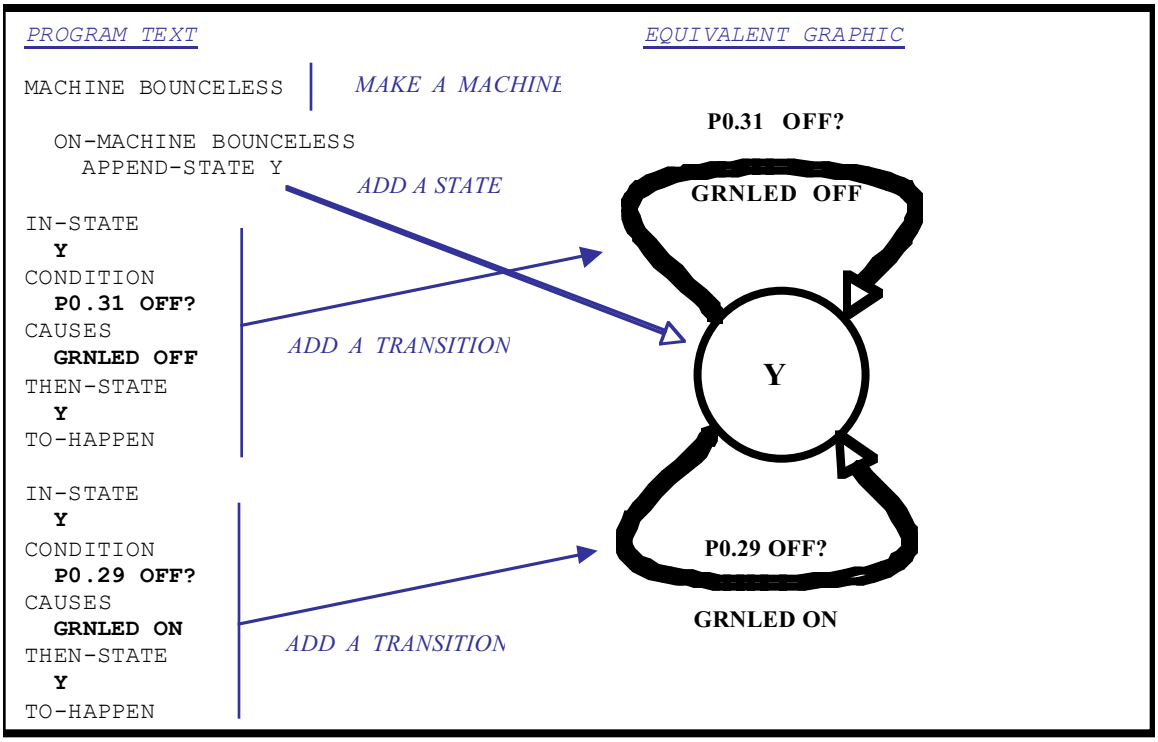
There you have yet another complete design, initialization and installation of a virtual machine in four lines of IsoMax™ code.

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

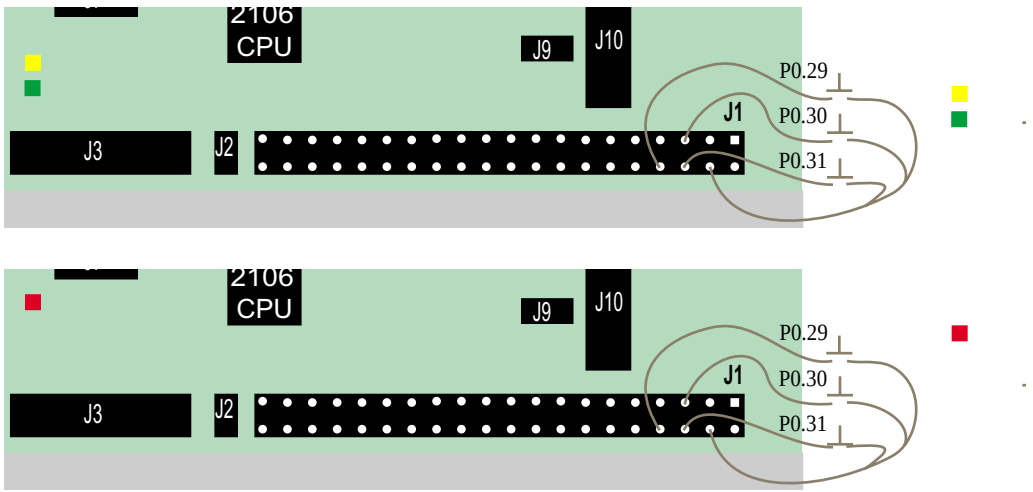


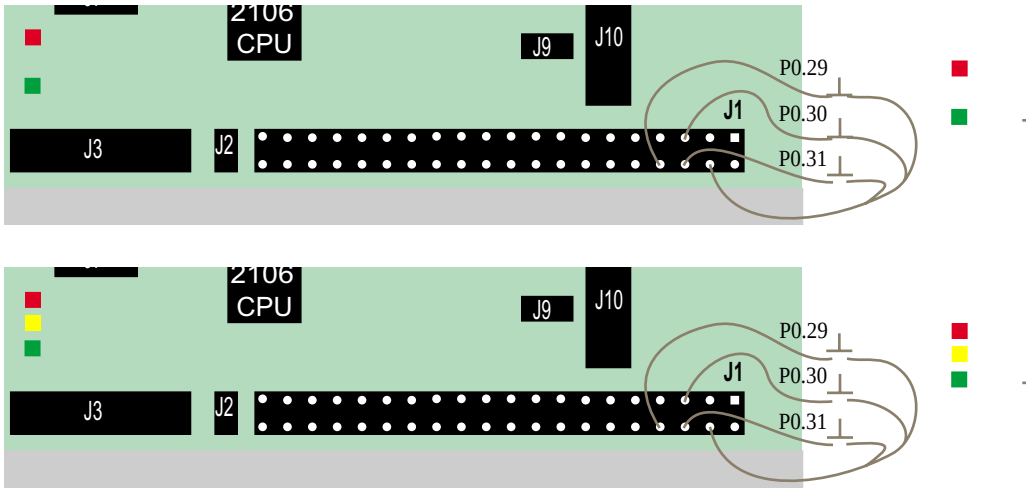
Bounceless switches filter out any noise on their input buttons, and give crisp, one-edge output signals. They do this by toggling state when an input first becomes active, and remaining in that state. If you are familiar with hardware, you might recognize the two gates feed back on each other as a very 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 P0.31 and P0.29 the green LED can be toggled from on to off with the press of the P0.31 button, or off to on with the press of the P0.29. The P0.31 button acts as a reset switch, and the P0.29 acts as a set switch.



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





Notice all three virtual hardware circuits are installed at the same time, they operate virtually in parallel, and the IsoMax™ 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.

4.4 SYNTAX AND FORMATTING

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

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

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

4.5 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.

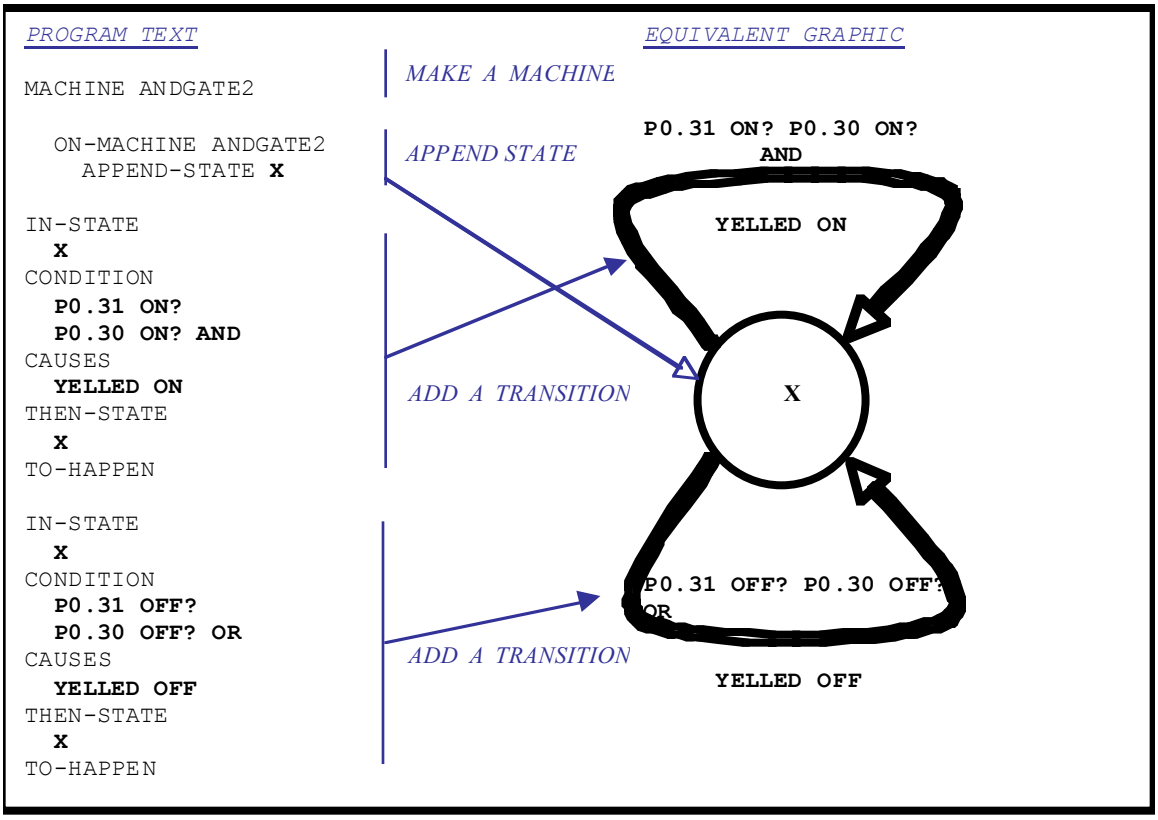
4.6 ANDGATE2

```
MACHINE ANDGATE2
  ON-MACHINE ANDGATE2
    APPEND-STATE X

IN-STATE
  X
CONDITION
  P0.31 ON?
  P0.30 ON? AND
CAUSES
  YELLED ON
THEN-STATE
  X
TO-HAPPEN

IN-STATE
  X
CONDITION
  P0.31 OFF?
  P0.30 OFF? OR
CAUSES
  YELLED OFF
THEN-STATE
  X
TO-HAPPEN

X SET-STATE  INSTALL ANDGATE2
```



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

TRANSITION COMPARISON		
ANDGATE1	**ANDGATE2**	
IN-STATE X CONDITION **YELLED OFF** P0.31 ON? P0.30 ON? AND CAUSES YELLED ON THEN-STATE X TO-HAPPEN	IN-STATE X CONDITION P0.31 ON? P0.30 ON? AND CAUSES YELLED ON THEN-STATE X TO-HAPPEN	IN-STATE X CONDITION P0.31 OFF? P0.30 OFF? OR CAUSES **YELLED OFF** THEN-STATE X TO-HAPPEN

The way this trick worked was by using an action in the condition clause, every time the

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

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

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

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

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

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

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

4.7 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 P0.31 ON? P0.30 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.

4.8 ANDGATE3

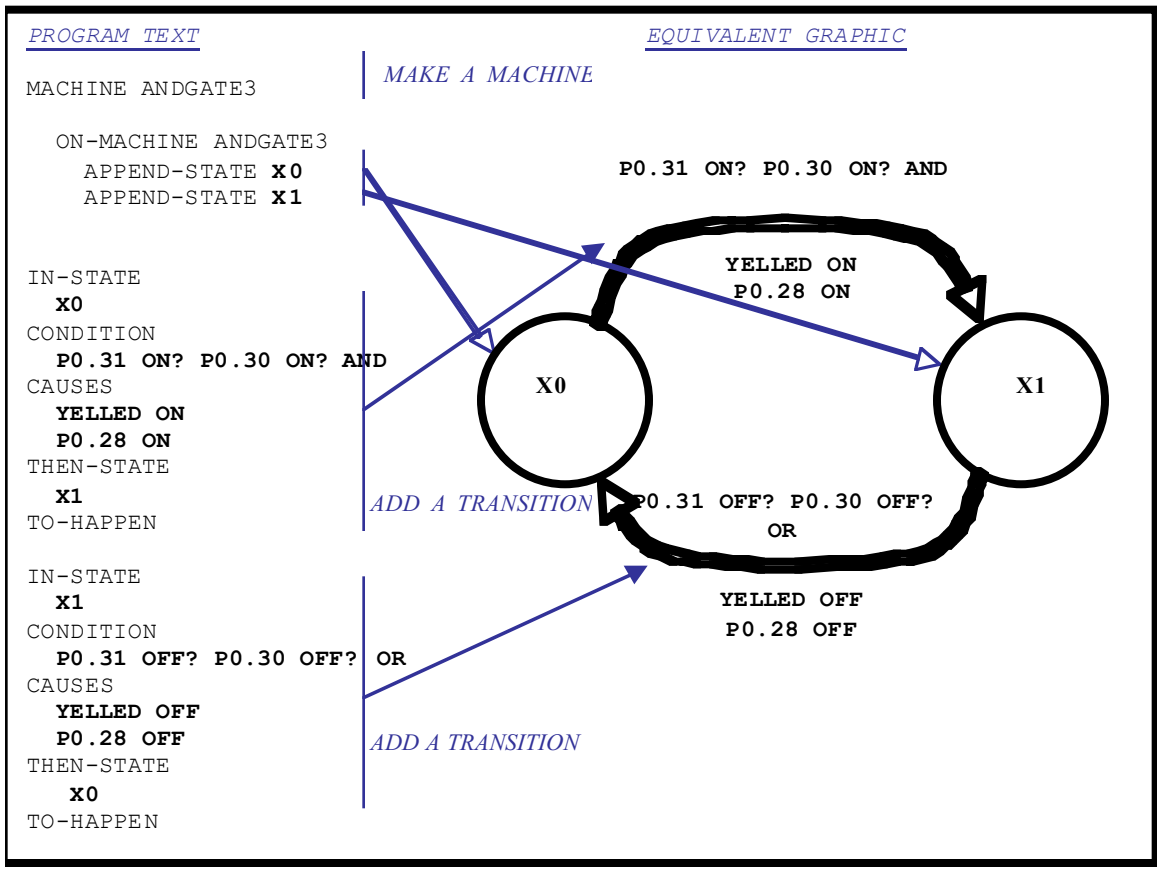
We are going to do another AND gate version, ANDGATE3, to illustrate this point about state machines having multiple states. This version will have two transitions and two states. Up until now, our machines have had a single state. Machines with a single state in general are not very versatile or interesting. 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.

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

```
  IN-STATE
    X0
  CONDITION
    P0.31 ON? P0.30 ON? AND
  CAUSES
    YELLED ON
    P0.28 ON
  THEN-STATE
    X1
  TO-HAPPEN

  IN-STATE
    X1
  CONDITION
    P0.31 OFF? P0.30 OFF? OR
  CAUSES
    YELLED OFF
    P0.28 OFF
  THEN-STATE
    X0
  TO-HAPPEN

  X0 SET-STATE  INSTALL ANDGATE3
```



Notice how similar this version of an AND gate, ANDGATE3, is to the previous version, ANDGATE2. The major difference is that there are two states instead of one. We also added some “spice” to the action clauses, doing another output on P0.28, to show how actions can be more complicated.

4.9 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 P0.31 and P0.30 are both high? If we had only one state, it would have to recalculate the AND phrase, or read back what ANDGATE3 had written as outputs. Rereading written outputs is sometimes dangerous, because there are hardware outputs which is cannot be read back. If we use different states for each different output, the state information itself stores which state is active. All an additional machine has to do to discover the status of P0.31 and P0.30 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 P0.31 and P0.30 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 P0.31 and P0.30 are both high.

4.10 STATE MEMORY

So you see, a state machine's current state is as much as an output as the outputs P0.28 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.

4.11 BOUNCELESS+

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

STATE Y	
IN-STATE Y CONDITION P0.31 OFF? CAUSES GRNLED OFF THEN-STATE Y TO-HAPPEN	IN-STATE Y CONDITION P0.29 OFF? CAUSES GRNLED ON THEN-STATE Y TO-HAPPEN

This code worked fine, as long as P0.31 and P0.29 were pressed one at a time. The green LED would go on and off without noise or bounces between states. Notice however, P0.31 and P0.28 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

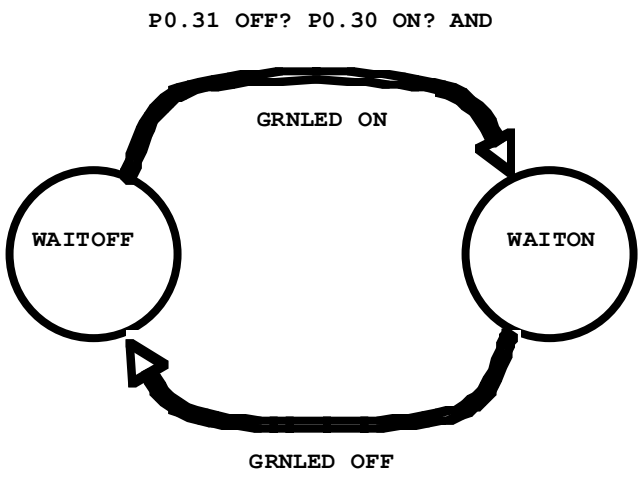
```
MACHINE BOUNCELESS+

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


IN-STATE
  WAITOFF
CONDITION
  P0.31 OFF? P0.30 ON? AND
CAUSES
  GRNLED ON
THEN-STATE
  WAITON
TO-HAPPEN

IN-STATE
  WAITON
CONDITION
  P0.30 OFF? P0.31 ON? AND
CAUSES
  GRNLED OFF
THEN-STATE
  WAITOFF
TO-HAPPEN
```

EQUIVALENT GRAPHIC



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

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

Further, the new BOUNCELESS+ machine is better behaved. (In fact, it is better behaved than the original hardware circuit shown!) It is truly bounceless, even if both switches are pressed at once. The first input detected down either takes us to its state or inhibits the release of its state. The other input can dance all it wants, as long as the one first down

remains down. Only when the original input is released can a new input cause a change of state. In the rare case where both signals occur at once, it is the history, the existing state, which determines the status of the machine.

STATE WAITOFF	STATE WAITON

IN-STATE
  WAITOFF
CONDITION
  P0.31 OFF? P0.30 ON? AND
CAUSES
  GRNLED ON
THEN-STATE
  WAITON
TO-HAPPEN
	IN-STATE
	  WAITON
	CONDITION
	  P0.30 OFF? P0.31 ON? AND
	CAUSES
	  GRNLED OFF
	THEN-STATE
	  WAITOFF
	TO-HAPPEN

4.12 DELAYS

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

Assembler	BASIC	C JAVA	FORTH
LOOP1 LDX # 0	FOR I=1 TO N	While (1)	BEGIN
LOOP2 DEX BNE LOOP2	LOOP2 DEX BNE LOOP2	{ delay(x);	DELAY
LDAA #1 STAA PORTA LDX # 0	LET PB=TRUE	out(1,portA1);	LED-ON
LOOP3 DEX BNE LOOP3	GOSUB DELAY	delay(x);	DELAY
LDAA #N STAA PORTA	Let PB=FALSE	out(0,portA1);	LED-OFF
JMP LOOP1	NEXT	}	AGAIN

Here's where IsoMax™ will start to look different from any other language you're likely to have ever seen before. The idea behind Virtually Parallel Machine Architecture is constructing virtual machines, each a little “state machine” in its own right. But this IsoStructure requires a limitation on the machine, themselves. In IsoMax™, there are no program loops, there are no backwards branches, there are no calls to time wasting delays

allowed. Instead we design machines with states. If we want a loop, we can make a state, then write a transition from that state that returns to that state, and accomplish roughly the same thing. Also in IsoMax™, there are no delay loops.

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

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

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

4.13 BLINKGRN

Now for an example of a delay using the count down to zero, we make a machine BLINKGRN. Reset your IsoMax™ 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 1- DUP , , DOES>
   DUP @ 0= IF DUP CELL+ @ 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

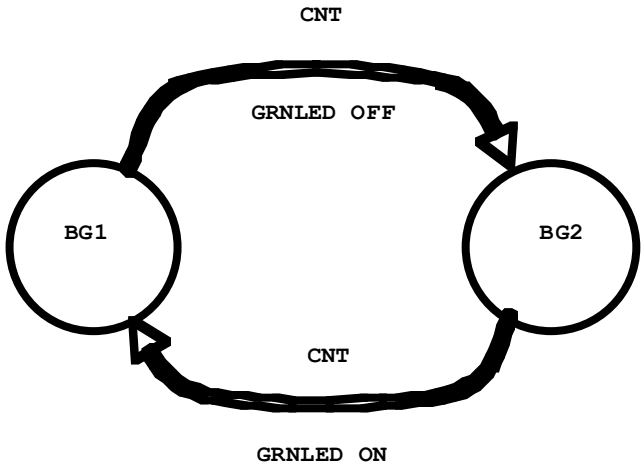
  ON-MACHINE BLINKGRN
    APPEND-STATE BG1
    APPEND-STATE BG2

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

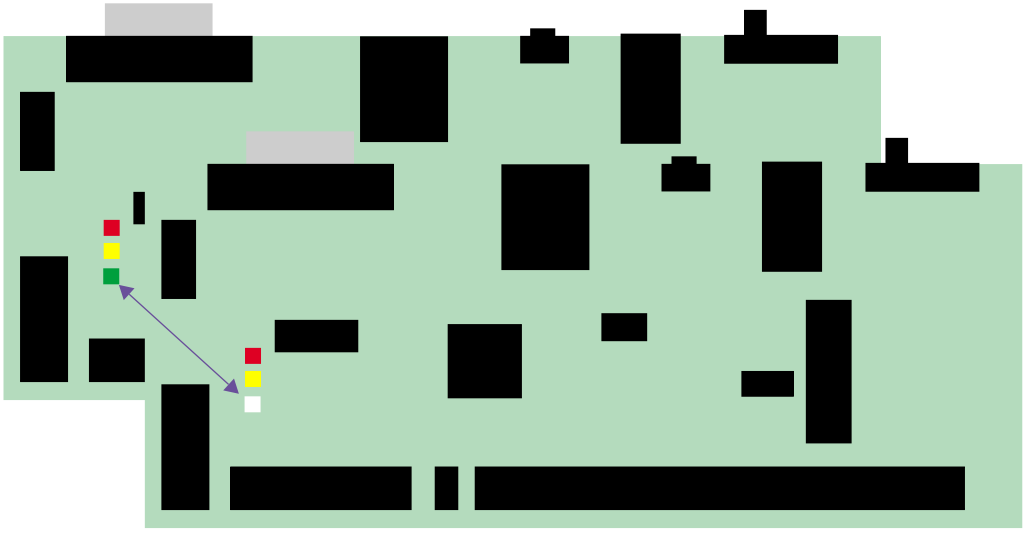
```
IN-STATE BG1 CONDITION CNT CAUSES GRNLED OFF THEN-STATE BG2 TO-HAPPEN

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
```

The result of this program is that the green LED blinks on and off. Every time the scheduler runs the machine chain, control is passed to whichever state BG1 or BG2 is active. The -LOOPVAR created word CNT is decremented and tested. When the CNT reaches zero, it is reinitialize back to the originally set value, and passes a Boolean on to be tested by the transition. If the Boolean is TRUE, the action is initiated.



The GRNLED is turned ON or OFF (as programmed in the active state) and the other state is set to happen the next control returns to this machine.

4.14 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 IsoMax™ can change state. First reset your machine to get rid of the existing machines.

4.15 ZIPGRN

```
MACHINE ZIPGRN

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

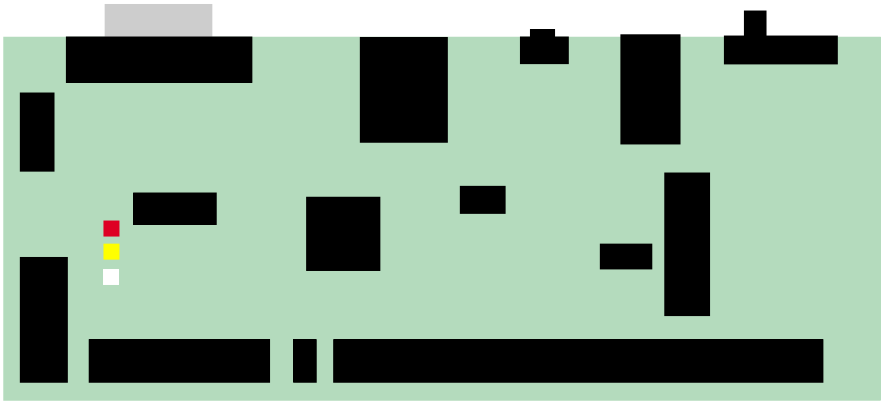
```
IN-STATE ZIPON CONDITION TRUE CAUSES GRNLED OFF THEN-STATE ZIPOFF
TO-HAPPEN

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

ZIPON SET-STATE
```

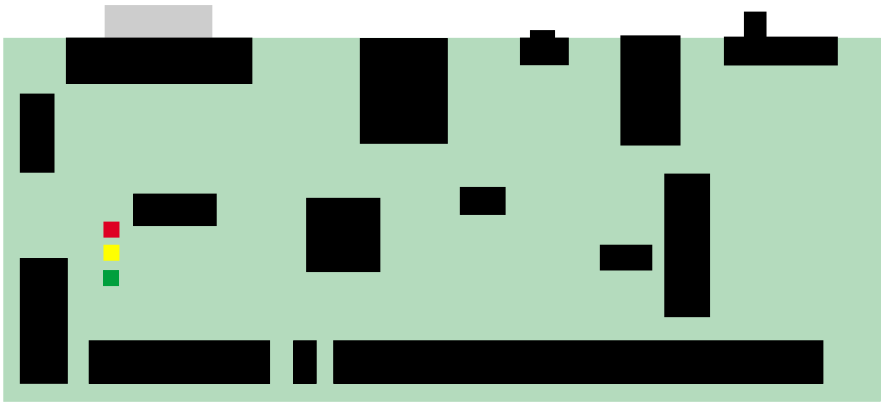
Now rather than install our new machine we're going to test it by running it “by hand” interactively. Type in:

```
ZPON SET-STATE
ZIPGRN
```



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

```
ZIPGRN
```



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

machine will change the green LED back and forth.

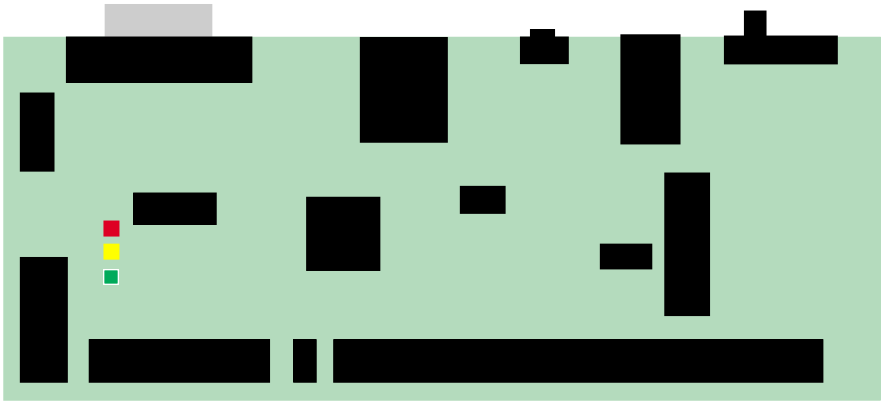
Now with the machine program and tested, we're ready to install the machine into the machine chain. The phrase to install a machine is :

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

So for our single machine we'd say:

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

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



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

4.16 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

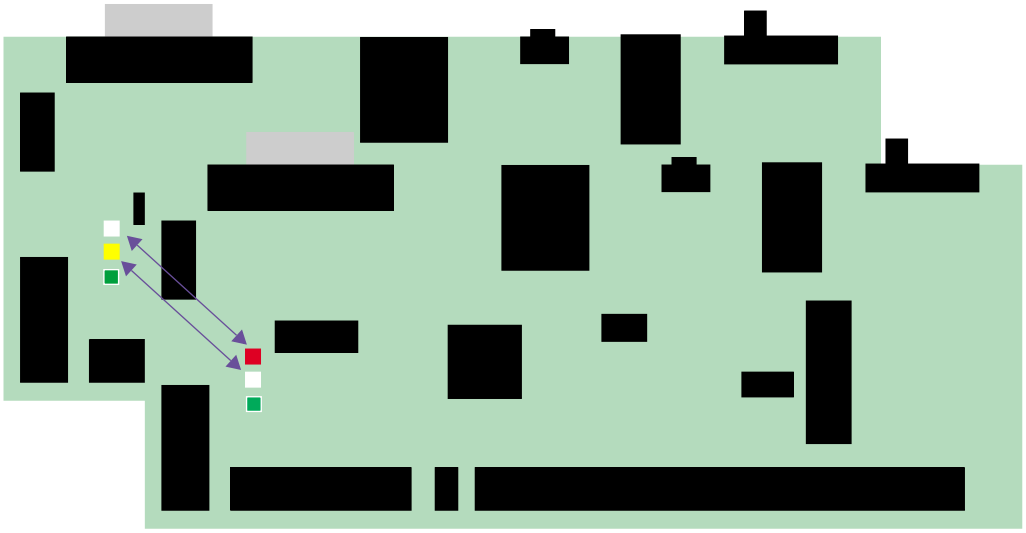
IN-STATE YELON CONDITION TRUE CAUSES REDLED ON YELLED OFF THEN-STATE
REDON TO-HAPPEN
```

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.

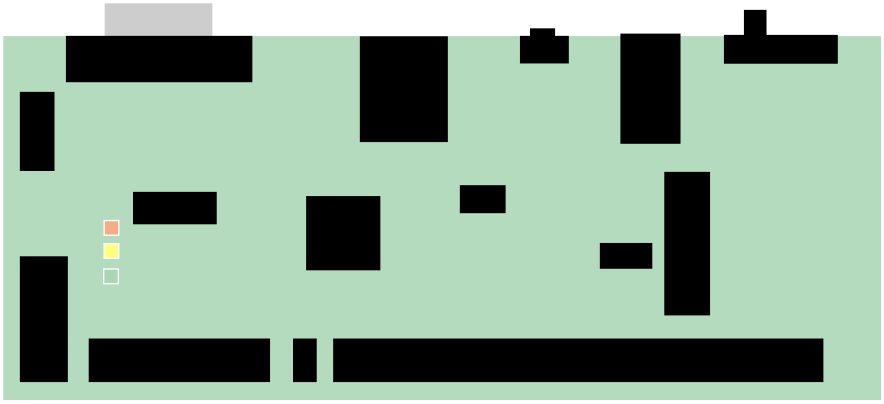


All this time, the ZIPGRN machine has been running in the background, because it is in the installed machine chain. Let's replace the installed machine chain with another. So we define a new machine chain with both our virtual machines in it, and install it.

```
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 IsoMax™.

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 100 MICROSECONDS SCHEDULE-RUNS CHN2
```

Let's see if we can press our luck.

```
EVERY 20 MICROSECONDS 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 IsoMax™ is running more than four hundred thousand high-level instructions per second. The IsoMax™ performance is unparalleled in any small computer available today.

4.17 TRINARIES

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

written with the trinary operators.

As an example, let's consider a digital thermostat. The thermostat works on a digital input with a temperature sensor that indicates the current temperature is either above or below the current set point. The old style thermostats had a coil made of two dissimilar metals, so as the temperature rose, the outside metal expanded more rapidly than the interior one, causing a mercury capsule to tip over. The mercury moving to one end of the capsule or the other made or broke the circuit. The additional weight of mercury caused a slight feedback widening the set point. Most heater systems are digital in nature as well. They are either on or off. They have no proportional range of heating settings, only heating and not heating. So in the case of a thermostat, everything necessary can be programmed with the machine format already known, and a digital input for temperature and a digital output for the heater, which can be programmed with trinaries.

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

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

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

```
HEX
DEFINE TOO-COLD? TEST-MASK 01 DATA-MASK 01 AT-ADDRESS 0FB1 FOR-INPUT
DEFINE TOO-HOT?  TEST-MASK 01 DATA-MASK 00 AT-ADDRESS 0FB1 FOR-INPUT
DECIMAL
```

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

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

A single output port line is needed to turn the heater on and off. The act of turning the heater on is unique and different from turning the heater off, however. Two actions need to be defined, therefore, even though only one I/O line is involved. P0.28 was selected for the heater control signal.

When P0.28 is high, or set, the heater is turned on. To make P0.28 high, requires P0.28 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 P0.28, 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 P0.28 is low, or clear, the heater is turned off. To make P0.28 low, requires P0.28 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 P0.28, is to be cleared. All other bits are to be left alone without being cleared.

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

```
HEX
DEFINE HEATER-ON  SET-MASK 02 CLR-MASK 00 AT-ADDRESS 0FB0 FOR-OUTPUT
DEFINE HEATER-OFF SET-MASK 00 CLR-MASK 02 AT-ADDRESS 0FB0 FOR-OUTPUT
DECIMAL
```

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

4.18 FLASH AND AUTOSTARTING

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

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

2. In the program file, each Forth word should be followed by FLWORD. 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* FLWORD (i.e., you must do IMMEDIATE FLWORD and *not* FLWORD IMMEDIATE).

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

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

6. Finally you can set the autostart vector in Program Flash. You need to provide an address on a 400h boundary, within unused Program Flash, thus after the end of the application program.

4.18.1 Autovectors:

By placing a tag at certain locations, the vector after it will be executed at one of three times: before startup (Quick) after startup (Boot) or before MaxForth runs (Autovector).

Quick tag is 0xA55A	0xFFF0
Quick vector is CFA at	0xFFF4
Boot tag is 0xA55A at	0xFFF8
Boot vector is CFA at	0xFFFC
Autovector range	0x8000 to 0x10004 at 0x400 boundaries: 0xA55A followed by CFA; default is at 0x10004

This simple program prints out a message when MaxForth starts up:

```
COLD
HEX FDP @ B000 FLERASE
: HI ." HELLO " ; FLWORD  HEX A55A FFF0 FL!  ' HI CFA FFF4 FL!
HI HELLO  OK
HELLO
IsoMax V1.0
```

Flash is erased first and then the startup program HI is stored in it. The boot vector is used as the startup hook. The first HELLO is from testing the code and the second one is from pressing the reset button which prints the start prompt as well.

4.18.2 Dimensions

Data stack and return stack are 64 cells deep. The floating point stack is 8 deep. Overflowing or underflowing any stack may result in unpredictable operation.

4.18.3 Updates

WORDS ("text" --) uses any following text as a search substring for selectively displaying only words which contain the text. If there is no following text, then all words are displayed. This is useful for listing subsets or finding particular words.

5. SOFTWARE

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 transports 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 discourages 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). Hire one single-minded watchman, and hire an executive to interrupt him if he becomes fixated, and move him along (multitasking). Hire one watchman who isn't so single-minded and can never be allowed to stop moving along on his rounds. (Isostructure)

5.1 WORD SYNTAX

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

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

AT-STATE <name-of-state> CONDITION <boolean computation... CAUSES <compound
AT-STATE <name-of-state> CONDITION <boolean computation... CAUSES <compound

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

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

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

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

EVERY <n> CYCLES SCHEDULE-RUNS ALL-TASKS
```

Under construction…

WITH-VALUE (-- 7100)stacks the tag 7100.

AT-ADDRESS (-- 7001)stacks the tag 7001. This will be topmost after ORDER.

AS-TAG (tag n tag n --)

Requires tags 7100,7001. Requires the latest word to be a State word. If it is, removes DUMMYTAG, 0 and replaces them with Address, Value.

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

Requires CSP=HERE. Requires the given word to be a State word. Then:

Removes last compiled cell. Compiles the CFA of the given State word. Compiles PTHIST.

NEXT-TIME (spfa --)

Requires CSP=HERE. Requires the given word to be a State word. Then:

Removes last compiled cell. Compiles the CFA of the given State word. Compiles PNEXTT.

SET-STATE (spfa --)

Given the pfa of a State word on the stack. Requires the given word to be a State word. Then:

Fetches the thread pointer and RAM pointer from the State word, and stores the thread pointer in the RAM pointer.

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

IN-EE

TIMING CONTROL

EVERY (-- 6000)stacks the value 6000.

CYCLES (-- 9000)stacks the value 9000.

SCHEDULE-RUNS not defined in source file

ALL-TASKS not defined in source file

COUNTDOWN-TIMERnot defined in source file

TIMER-INIT not defined in source file

INPUT/OUTPUT TRINARIES

DEFINE <word-name>(-- 1111)

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

PROC not defined in source file

END-PROC not defined in source file

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

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

FOR-INPUT (1111 tag n tag n tag n --)

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

XCPAT

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

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

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

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

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

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

5.2 Queues

Queues are for when you don't have time to wait. If data could be used before more data was passed along, then data would flow unhindered. As it really is, often we must wait. In a traditional sense this would be a loop checking for a resource to become available before transferring data. In an IsoMax state machine, a state would have an event for checking the resource availability. But what if we wanted to batch up our processing and move a bunch of data at once? This requires a memory structure and management tools which are described in this section. Queues are a natural extension of data flow in that they are data stopped but ready to go.

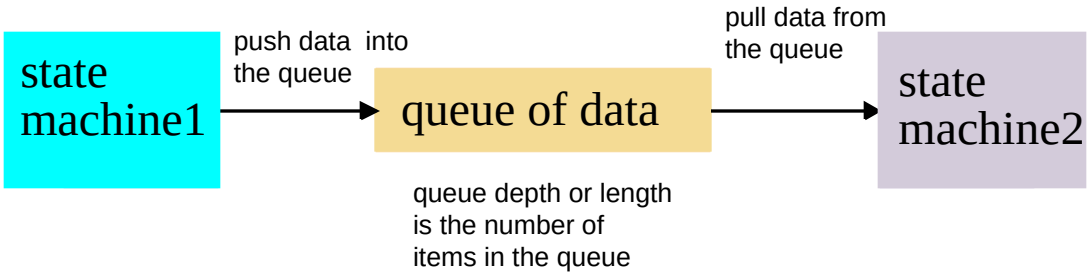


Figure 1 State machine 1 and state machine 2 have a form of elasticity between them. If state machine 1 has data for state machine 2, it does not have to wait for state machine 2 to be available

Data is pushed into a queue at one end when ready and then pulled out at the other end when ready to be processed. As long as the overall flow in is less than the overall flow out, then the queue will never fill up or overflow. The length of a queue is related to the maximum difference between inflow and outflow at any point in time. The length of queue can compensate for irregular data flow.

For instance say we connect two different serial ports with port A at 9600 and port B at 115200 or 12 times faster. Now data flowing from A to B will never have a problem and a queue between the two state machines for the two ports can be quite small or maybe even just a variable. However in the other direction, B can bring data in at a rate 12 times faster than A can process it. So a queue between the state machien for port B and the one for port A would have to be big enough to hold on to a burst of data. And the maximum average throughput on B would be 12.5%. In the case that bursts of 16 came in, then the queue would have to be 16 or more cells long to hold the stopped data.

5.2.1 Queues for supporting data and control flows

Queue structure: | insert | remove | end | data space for circular queue |

```
    +-----------------------------+
    |         +--------------+    |
    |         |              V    V
| insert | remove | end |  data cells... | last data cell |
                      |                  ^
                      +------------------+
```

```
PUSH --> |  QUEUE   | --> PULL
POP  <-- |          | <-- STUFF
```

The first cell is the insert pointer; the second cell is the removal pointer and the third cell is the end pointer. Both insert and removal pointers are respectively moved down in memory by PUSH and PULL. POP and STUFF move them up in memory. When either pointer gets to the pointer for end or the end pointer, if going the other way, they are set to the end pointer or just after the pointer to the end pointer. When the queue is zeroed, both pointers are set to the same location. Since it contains nothing, the insert pointer always points to a location where values can be written. Once a value has been written with PUSH, then it is to point to another empty location. The removal pointer points to the value to be pulled and when PULL is called, it is decremented. POP uses the insert pointer and STUFF uses the removal pointer and they go the other way. The structure of the queue is most efficient for the use of

PUSH, PULL and Q. Also if you ask for n cells in the queue, you actually get n+1. This serves as an overflow for the case when you say 0 QUEUE and it also allows the queue to contain n items before it is full as opposed to n-1 items.

TABLE 4. Queue pointers used by operation

action	i	r	e
PUSH	*		*
POP	*		*
PULL		*	*
STUFF		*	*
Q		*	
P	*		
Q?	*	*	*
QSIZE			*
0Q	*	*	*

For high speed data flow, memory could be accessed from a queue engine in hardware for the `i,r,e` pointers.

5.2.2 Queues as a component of Data Flow

By specifying the width and depth of a queue, certain values correspond to certain objects:

TABLE 5.

width	depth	Object
0	0	no connection, and no hardware is needed
1	0	wire and is just a connection
2 or more	0	bus
1	1	flip flop
2 or more	1	register
1	2 or more	shift register
2 or more	2 or more	queue

This suggests a general architectural component with specifiable width and depth where the proper subcomponent is chosen based on the above objects for hardware implementations, and the user model is just a line on a diagram or segment of script. The low level architectural building blocks are queues and machines.

5.2.3 Word Definitions

QUEUE (n “name” --) create a queue in memory with n cells and name

0Q(q --) remove all items from a queue by equating insert and removal pointers

Q?(q -- n) return number of items in a queue

PUSH (n \ q --) append n to end of queue

PULL (q -- n) return first value in queue

POP (q -- n) return last value in queue

STUFF (n \ q --) prepend to beginning of queue

P (q -- n) get a copy of last value in queue

Q(q -- n) get a copy of first value in queue

QLEFT (q -- n) return number of cells left in queue

QSIZE (q -- n) return the size of a queue as allocated at compile time

RESIZEQ (n \ q --) change queue size by moving end pointer (cannot be made bigger than original space)

ROTATEQ (n \ q --) rotate n items from head of queue to end of queue

TRANSFERQ (qa \ qb \ n --) transfer n items from head of qa to tail of qb

5.3 Timeouts

TImeouts are a resource used to measure time periods either for a single or cyclic purpose. The base time unit is one microsecond and on a 32-bit architecture, this give a range of up to 71 minutes. As many timeouts that are needed may be declared. Note that if you change the period with PERIOD or with SCHEDULE-RUNS, then this changes the time base so all timeouts will be invalid.

5.3.1 Word Definitions

GET-TIME (-- t) return current raw time value

MICROSECONDS (n -- t) convert n to microseconds raw time

MILLISECONDS (n -- t) convert n to milliseconds raw time

SECONDS (n -- t) convert n to seconds of raw time

MINUTES (n -- t) convert n to minutes of raw time

SET-TIMEOUT (t \ timeout --) set the new timeout time and start it

RETIMEOUT (timeout --) restart a timeout based on a multiple of the start time where period is exact

START-TIMEOUT (timeout --) start a timeout from right now or for aperiodic timeouts

TIMEOUT (t \ “name” --) declare a timeout timer for use in state machines

TIMEOUT? (timeout --) return flag indicating a timeout is done or not

5.4 Other Definitions

These are extra definitions present on the ARM IsoMax:

DINT (--) disable all interrupts

EINT (--) enable all interrupts

S (--) a download program for text and s-records with no echoback. Exit with QUIT

skips (-- a) variable for the number of times that the SSM machine has been blocked because a SM was taking longer than a SSM period to execute

stackerror (-- a) variable set when the stacks used by a machine are not empty when done

emitv (-- a) vector for intercepting emited values. If emit character is changed to zero, then no character is emitted. Used by S.

pasm (-- a) points to the beginning of the ARM exception vectors where asynchronous state machines are placed.

5.5 ASMs and SSMs

This release adds and updates some tools.

.MACHINES lists all the active machines, syncnronous (SSM) and asynchronous (ASM)

.STATES lists all the active states:

Example usage:

```
.STATES Active states: HT-IDLE NOT-GRAPHING  OK

.MACHINES
SSMs: HT-POLL GRAPHER
ASMs: LCD-TRANSFER  DATA-TRANSFER  OK
```

The ASMs associate with a peripheral like a timer, uart or pwm so I'm thinking of adding that to the printout so that you know which ASM is assigned where.

Examples of ASM assignment where `USE-ASM` is like `INSTALL` but takes an interrupt source:

```
  TIMER0 USE-ASM LCD-TRANSFER
  UART1 USE-ASM DATA-TRANSFER
```

Reseting the interrupt: If the user is using a timer, then resetting the interrupt will depend on the channel and also on the usage. The system cannot know what to reset, so the user must reset the interrupt generated by the peripheral. However, the interrupts on the ARM are handled by a complex beast known as the VIC (vectored interrupt control). The ASM interface provided basically hides the VIC and its complexity. The VIC also has to have its interrupt reset but that is always the same so the system takes care of it. This only leaves the user with the peripheral they want to use and its registers to deal with. A couple of exampls of ASM's, SSM's, timeouts and queues are included for reference.

This is an example of an SSM used to obtain readings from a sensor:

```
\ humidity and temperature machine
MACHINE HT-POLL
  ON-MACHINE HT-POLL
    APPEND-STATE SHT-OFF  \ 11 ms before awake
    APPEND-STATE SHT-WAKING
    APPEND-STATE HT-IDLE
    APPEND-STATE HO-HUM
    APPEND-STATE TEMP-WAIT

IN-STATE SHT-WAKING
  CONDITION sht-to TIMEOUT? CAUSES
    ht-to START-TIMEOUT
  THEN-STATE HT-IDLE TO-HAPPEN

IN-STATE TEMP-WAIT
  CONDITION DAT-READY? CAUSES
    GET-DAT rawtempq PUSH
    5 SHT-CMD
  THEN-STATE HO-HUM TO-HAPPEN

IN-STATE HO-HUM
  CONDITION DAT-READY? CAUSES
    GET-DAT rawhumq PUSH
    SHOW-TERH
```

```
  THEN-STATE HT-IDLE TO-HAPPEN

IN-STATE HT-IDLE
  CONDITION ht-to TIMEOUT? CAUSES
    3 SHT-CMD  ht-to START-TIMEOUT
  THEN-STATE TEMP-WAIT TO-HAPPEN

IN-STATE HT-IDLE
  CONDITION graph-to TIMEOUT? CAUSES
    graph-to START-TIMEOUT  GRAPH-TERH
THEN-STATE HT-IDLE TO-HAPPEN
```

This is an example of an ASM used to control a timer reading a sensor on two channels. The sample window is set by a parallel SSM which runs periodically while the ASM handls the asynchronous nature of the timer channel edge interrupts. Note the use of the debug tool `LEDS` to show which state the machine is in when a crash happens and as a bonus, it looks pretty when it runs:

```
HEX
MACHINE 2D-TILTS
  APPEND-STATE LR-SYNCING
  APPEND-STATE LR-RISING
  APPEND-STATE LR-FALLING
  APPEND-STATE LR-PERIOD
  APPEND-STATE FA-SYNCING
  APPEND-STATE FA-RISING
  APPEND-STATE FA-FALLING
  APPEND-STATE FA-PERIOD
  APPEND-STATE TILT-OFF

IN-STATE LR-SYNCING
  CONDITION 1 CAUSES 1 LEDS
    TIMER1_CR0 @ DROP  \ toss possibly stale data
  THEN-STATE LR-RISING TO-HAPPEN

IN-STATE LR-RISING
  CONDITION 1 CAUSES 2 LEDS
    TIMER1_CR0 @ start-count !
    0 TIMER1_CCR !  6 TIMER1_CCR !
  THEN-STATE LR-FALLING TO-HAPPEN

IN-STATE LR-FALLING
  CONDITION 1 CAUSES 3 LEDS
    TIMER1_CR0 @ start-count @ -
    DUP lr-period @ >  IF  lr-period @  MOD  THEN
    lrq PUSH
    0 TIMER1_CCR !  5 TIMER1_CCR !
  THEN-STATE LR-PERIOD TO-HAPPEN

IN-STATE LR-PERIOD
  CONDITION 1 CAUSES 7 LEDS
    TIMER1_CR0 @ start-count @ - lr-period !
    0 TIMER1_CCR ! 28 TIMER1_CCR !
  THEN-STATE FA-SYNCING TO-HAPPEN

IN-STATE FA-SYNCING
```

```
  CONDITION 1 CAUSES 4 LEDS
    TIMER1_CR1 @ DROP
  THEN-STATE FA-RISING TO-HAPPEN

IN-STATE FA-RISING
  CONDITION 1 CAUSES 5 LEDS
    TIMER1_CR1 @ start-count !
    0 TIMER1_CCR !  30 TIMER1_CCR !
  THEN-STATE FA-FALLING TO-HAPPEN

IN-STATE FA-FALLING
  CONDITION 1 CAUSES 6 LEDS
    TIMER1_CR1 @ start-count @ -
    DUP fa-period @ >  IF  fa-period @ MOD  THEN
    faq PUSH
    0 TIMER1_CCR !  28 TIMER1_CCR !
  THEN-STATE FA-PERIOD TO-HAPPEN

IN-STATE FA-PERIOD
  CONDITION 1 CAUSES 0 LEDS
    TIMER1_CR1 @ start-count @ - fa-period !
    0 TIMER1_CCR !
  THEN-STATE TILT-OFF TO-HAPPEN

\ One timer, two interrupt sources so this SM is a demux
: TILTS  ( -- )  TIMER1_IR @ TIMER1_IR !  2D-TILTS ;

: QS DUP RIQ DUP QSIZE 0 DO DUP RPOP . LOOP DROP ;

DECIMAL
100 MILLISECONDS TIMEOUT stample-to \ 100ms sample period

HEX
: SAMPLE-TILT  ( -- )  stample-to TIMEOUT?
   IF  stample-to RETIMEOUT  LR-SYNCING SET-STATE
       5 TIMER1_CCR !  \ interrupt on each rising edge of ch 0
       TIMER1_IR @ TIMER1_IR !  \ clear any pending ints
   THEN ;

: INIT-TILT  0 LEDS  lrq 0Q  faq 0Q
   A00000 PINSEL0 @ OR PINSEL0 !
   TILT-OFF SET-STATE
   TIMER1 USE-ASM TILTS  \ assign the ASM to timer 1 interrupt
   1 TIMER1_TCR !  \ enable counter 1
   stample-to START-TIMEOUT  INSTALL SAMPLE-TILT ;
```

Other useful tools include:

INTS - list all interrupts pending. If used with DINT and EINT, the running interrupts can be determined.

NO-ASM (interrupt --) disconnect the ASM from that interrupt

.H (n --) print a number in unsigned hex irrespective of base

As for the diagrams of state machines, for the interested, there is a document covering state machine diagrams and fitting them in the bigger picture (literally) for a project for those that are diagram or visually oriented:

http://www.ee.ualberta.ca/~rchapman/Cmpe401/pdfs/BubbleDiagramStratification.pdf

If your documenter can click into parts of the diagram, all the better, it can help organize it hierarchially.

5.6 More Tricks

To stop the Forth Interpreter:

```
VARIABLE running  0 running !
: HALT  BEGIN  running @  UNTIL ;
HALT
```

This will stop the Forth interpreter from doing anything until a part of your program changes the state of the variable. This allows two communicating state machines to signal asynchronously. `HALT` is a simple one line state for the interpreter and your code would be the other. The interpreter inputs and outputs characters via queues. These are like the variable running but deeper in that one state machine can store multiple times to the queue and another can read multiple times from the same queue without contention. This is achieved with two pointers and a single writer. The queues for KEY and `EMIT` are filled either by asynchronous state machines (ASM) triggered by the UART interrupt or by `KEY` and `EMIT` if the sio needs servicing and interrupts are off. This way it works with interrupt on or off. Currently there are no interrupts feeding the sio and you can verify this by reading `UART0_IER`. So by putting the interpreter into the `HALT` state, no uart accesses will hamper your code which runs as an SSM or ASM.

HALT is a word I created to halt the interpreter or from the user point of view, lock up. The interpreter is the main thread of code running from bootup but with IsoMax it is not the only threads of code able to run. IsoMax runs the state machines that you program as a thread of code from an interrupt. So while the interpreter is running continuously or halted, the state machines will still run. Just stopping the interpreter prevents it from grabbing input or creating output. The interpreter can be unhalted by changing the value of the blocking variable running in a state machine. When you INSTALL a state machine in IsoMax, you are making it part of a set of state machines that are EXECUTEd every PERIOD by an interrupt from timer 0 match register 3. You can disable and enable interrupts with DINT and EINT. In IsoMax for ARM, the other interrupts for the other peripherals on the ARM are available to execute state machines as well. This really lets loose the power of the micro yet orchestrates it perfectly.

Probably the missing info is where HALT gets called from. It must be typed in and intepreted from the command line or it could be run at the end of startup say for a small window to hit a key upon reboot to gain control of a serial port.

As for `PERIOD`, the synchronous state machines (SSM) are run from a single clock edge like hardware state machines. The period of the clock can be set to as low as 100 or 50 microseconds depending on what you need to do. Any lower than this and it takes the processor more time to execute the code than a period and the interpret never gets to run. The default period is 1 millisecond. The settable time modifers are `MINUTES`, `SECONDS`, `MILLISECONDS` and `MICROSECONDS`. These are also use with the timeouts. The SSMs are clocked by timer0 match register 3.

To run a state machine as an SSM you use `INSTALL`. To run an ASM it is a little more involved and is covered in the manual. For the next release, there should be some simplification and VIC automation to make it at easy as `INSTALL`.

Another way to disconnect the interpreter from the UARTs is to point the sio pointers at a set of fake registers in RAM. You could even use the RAM registers to communicate with the interpreter by simulating the UART register bits. This could be done from a state machine.

To have three uarts, you can simulate a low baudrate using a general IO pin and toggling it to simulate a UART.

5.7 From Loops to States

This is a refactoring experience that occurred while building one of the examples which has some merit.

Initially I had two state machines, one for each signal running from the same interrupt source but on

different channels. I was getting a lot of noise and bad signals but I knew it wasn't the signal, so ai needed a new refactoring of the solution. Architecturally, I merged two state machines into one so that they can run sequentially to reduce contention. Also since there is only one interrupt vector (TIMER1), it maps better to a single state machine. The states from the Left-Right state machine and the Fore-Aft state machine now combine to become the 2D-TILTS state machine. This also makes it easier to coordinate common states like powerup, idle and powerdown. The tilt sensor states are:

```
MACHINE 2D-TILTS  \ capture and measure tilt signals
  APPEND-STATE LR-SYNCING  \ find a rising edge on first signal
  APPEND-STATE LR-RISING   \ record a rising edge
  APPEND-STATE LR-FALLING  \ record pulse length
  APPEND-STATE LR-PERIOD   \ record period length
  APPEND-STATE FA-SYNCING  \ repeat for other signal
  APPEND-STATE FA-RISING
  APPEND-STATE FA-FALLING
  APPEND-STATE FA-PERIOD
  APPEND-STATE TILT-OFF    \ no measurements; can be powered off
```

The syncing state waits for a rising edge and then discards the reading. The rising and falling states record and subtract the times to get the pulse width and then the period state takes a final rising edge measurement to gauge the period for that pulse. One signal is dealt with, then the other, changing edge triggers each state in TIMER1_CCR always writing zero in between. This worked quite well and I could trigger a reading when needed except sometimes the pulse and period would be one period longer. I cleaned up the glitch by using the previous period and modulus but that got me to digging for the source of the glitch.

There are a total of 6 interrupts running on the the 2106, 4 synchronously, 2 asynchronously. I used the TIMER_TC to time how long it takes to service each interrupt and then keep the maximum time value. Then I view these values to see how long the interrupts were taking. It turns out that occasionally, one interrupt, TILT-VIEWER which did LCD graphing was taking 1.5ms, which was longer than the LR or FA periods (1ms). So I refactored the looping code from inside one state into multiple states which displayed the current value, and then graphed the historical value, for both signals sequentially and synchronously. The tilt viewer state are:

```
MACHINE TILT-VIEWER  \ View tilt signals on LCD
  APPEND-STATE TILT-HIDE     \ don't show the tilt signals
  APPEND-STATE TILT-SHOW     \ add LCD window dressing for signals
  APPEND-STATE TILT-UPDATE   \ update LCD tilt display periodically
  APPEND-STATE LR-CURRENT    \ display current left right signal value
  APPEND-STATE LR-HISTORY    \ graph last 11sec of left right signal
  APPEND-STATE FA-CURRENT    \ display current fore-aft value
  APPEND-STATE FA-HISTORY    \ graph last 11sec of fore-aft values
```

Once I started slicing, dicing and synchronizing the smaller components, everything ran a whole lot smoother and faster. So far so good. Now I will attempt to get nested interrupts running which are in the system but seem to be broken which is what got me into this tool building and refactoring phase in the first place. There is no end in sight if the journey is interesting.

The best thing I liked doing in hardware design was the state machines but the hardest part was getting them working just right with the least logic. Now I find with coding a real time embedded system, the state machine factoring gives me the power to create a much finer designed system, only with software it is much easier to change. Its like having a better harness on the whole program code structure.

5.8 Tools

5.8.1 Autoboot

RAM-START (-- a) start address of dictionary in RAM

BOOT-VECT (-- a) start address in flash for image

RESTORE (--) word used to restore RAM from ROM at boot

BOOTUP (tick --) used to save RAM image to ROM and call
tick word after restoring image at boot

-AUTOBOOT (--) turns off the autoboot flash image

With software installed in flash for autoboot, each time a reset is done, RAM will be refreshed with the contents of ROM and any thing extra compiled in RAM is gone. To modify this behaviour, use -AUTOBOOT to turn off the autobooting and then a warm image will be kept through reset but the files will have to be downloaded each time a power cycle is done. The other option is to compile your code on top and then save it to ROM using:

```
' QUIT BOOTUP
```

After a reset this will leave a freshly compiled image in RAM from ROM but it doesn't run any of the software so it is just like it was after compiling. This is handy if you usually work on one file for a while and all the files before it are handy to keep in ROM. Then you can just press the reset button and drag and drop the one file you are testing (with embedded test code) and you don't even have to leave the editor application.

5.8.2 Text:

CELL (-- n) returns the number of bytes in a cell

.H (n --) prints n as an unsigned hex number

` ('C' --) gets ASCII value of next character

" ("s" -- s) create a count prefixed string with "

5.8.3 Debugging:

LEDS (n --) 0-7 selects one of 8 led states: 0 all off; 1 green; 2 yellow; 3 yellow , green; 4 red; 5 red, green; 6 red, yellow; 7 red, yellow, green

badints (-- a) incremented for each nonassigned interrupt in the VIC.

5.8.4 Asynchronous State Machines:

WDT, SWI, DBRX, DBTX, TIMER0, TIMER1, UART0, UART1, PWM0, I2C, SPI, PLL, RTC, EINT0, EINT1, EINT2 (-- i) interrupt number for ASMs

USE-ASM (i \ "sm" --) assign a state machine to an interrupt n

NO-ASM (i --) turn off interrupt

NO-MACHINES (--) stop all state machines, ASMs and SSMs

5.8.5 Status:

INTS (--) pending system interrupts

.MACHINES (--) lists all running SSMs and ASMs

.STATES (--) lists all the current states for all state machines

5.8.6 Measurement:

START (-- n) time in microseconds

END (n --) shows elapsed time since START

ENDMAX (n \ a --) keeps maximum elapsed time at address a

STARTMAX ("sm" -- sm \ t) leave START and state address

STATEMAX (sm \ t \ v --) store state and time in v is maximum

5.9 ASM Example: Stoplight

Here is a simple ASM example using timer 1 to treat the onboard leds, using LEDS tool, to make them

run like a stoplight:

```
\ Stoplight demo  Rob Chapman  Feb 3, 05

\ This example demonstrates using an ASM with the TIMER1 peripheral
\ Each interrupt, the timeout length is changed and an LED is set
\ green for 5 seconds; yellow for 1 second; red for 5 seconds

MACHINE STOPLIGHT
  APPEND-STATE GREEN
  APPEND-STATE YELLOW
  APPEND-STATE RED

IN-STATE GREEN  \ light is green till timeout happens
  CONDITION 1 CAUSES
    2 LEDS  1 TIMER1_MR0 !  1 TIMER1_IR !  \ change to yellow for one second
    1 3 TIMER1_TCR !  TIMER1_TCR !  \ since TC doesn't reset with smaller MR
  THEN-STATE YELLOW TO-HAPPEN

IN-STATE YELLOW  \ light is yellow till timeout happens
  CONDITION 1 CAUSES
    4 LEDS  5 TIMER1_MR0 !  1 TIMER1_IR !  \ change to red for 5 seconds
  THEN-STATE RED TO-HAPPEN


IN-STATE RED  \ light is red till timeout happens
  CONDITION 1 CAUSES
    1 LEDS  5 TIMER1_MR0 !  1 TIMER1_IR !  \ change to green for 5 seconds
  THEN-STATE GREEN TO-HAPPEN

: RUN-STOPLIGHT
   3 TIMER1_TCR !  1 TIMER1_MR0 !  60000000 1-  TIMER1_PR !  \ 1 second base
   3 TIMER1_MCR !    1 TIMER1_IR !
   0 LEDS  YELLOW SET-STATE  TIMER1 USE-ASM STOPLIGHT  1 TIMER1_TCR ! ;

RUN-STOPLIGHT

\ Note: when using a long prescale count, the interrupt can be serviced and
\ over with before the mr count is reset.  When changing the MR to a smaller
\ value, the counter doesn't get reset to zero even though the MCR is set to
\ make it do so.  Work around is to either use multiple MRs or reset counter
\ manually.
```

One of the more useful tools for tight real time software is a measuring tool to see how long things, usually states, take to execute. This can help to refactor a system to make it very efficient. The measurement tools can be applied to find out which state takes the longest to execute and how long that is by adding in the following code before RUN-STOPLIGHT:

```
\ Measurements
CREATE slmax 0 , 0 ,
: STOPLIGHT STARTMAX STOPLIGHT STOPLIGHT slmax STATEMAX ;
: 0MAXS  0 slmax ! ;
: MAXS   slmax @ U. ." us in state: " slmax CELL+ @ .STATE ;

: RUN-STOPLIGHT ...
```

Now we can let it run and then examine the timings:

```
MAXS 38 us in state: YELLOW  OK
MAXS 41 us in state: GREEN  OK
```

This was done before a complete cycle had been done so yellow is shown first and then the next query shows green. The longest state is GREEN and this makes sense since it has the most code. This tool can be applied to determine which states need refactoring.

5.10 Word Listing (LPC2106)

```
IsoMax for ARM Beta 5
WORDS
40001EB8 ISOMAXBETA       40001E2C INIT             40001DB0 ALL-MACHINES
40001D80 +SSMS            40001D58 -SSMS            40001D40 ssmblock
40001D24 ssmmax           40001D00 EIRQ             40001CD4 STARTMAX
40001CB0 'CURCOND         40001C4C STATEMAX         40001C0C ENDMAX
40001BD8 END              40001BBC START            40001B74 .CURSTATE
40001A9C .STATE           400019F0 .STATES          400019D8 XSTAT
400019C0 DTAG             40001998 'COND            40001970 'DTAG
40001950 'STATE           40001934 'PARENT          4000186C INTS
400017FC NO-MACHINES      40001758 .MACHINES        40001720 .H
400016B8 CFA>NFA          400015FC NO-ASM           400015B0 USE-ASM
400014C4 (USE-ASM)        400014A8 EINT2            40001490 EINT1
40001478 EINT0            40001460 RTC              4000144C PLL
40001438 RESERVED         4000141C SPI              40001408 I2C
400013F4 PWM0             400013DC UART1            400013C4 UART0
400013AC TIMER1           40001394 TIMER0           4000137C DBTX
40001364 DBRX             4000134C SWI              40001338 WDT
40001300 BADINTS          400012E8 badints          40001274 LEDS
40001240 LEDS-ON          400011C0 "                40001198 `
4000116C -AUTOBOOT        40000FF4 BOOTUP           40000F60 RESTORE
40000F48 BOOT-VECT        40000F24 NEXT-TIME        40000F00 THIS-TIME
40000EDC TO-HAPPEN        40000EAC FLSTATE          40000D30 IN-EE
40000D08 CONDITION        40000CEC LAST-CONDITION   40000CC4 APPEND-STATE
40000C88 MACHINE          40000C30 UNDO             40000B5C FORGET
40000B3C :                40000B18 VARIABLE         40000AD4 CREATE
40000ABC RAM-START        40000A94 CONSTANT         40000A4C ?ALIGN
40000A34 CELL             40000A14 END-CODE         400009F0 ;
400009D8 [IMMEDIATE]      400008B8 FLWORD               155C TASK
     A10 (                    27AC @                   2750 C@
    285C !                    2754 C!                   33C 2@
     32C 2!                   1878 :                   189C ;
    2814 +                    27F0 -                    110 1-!
      FC 1+!                  281C +!                  2858 *
     134 /                    2868 ><                  2864 SWAP
     37C 2OVER                 364 2SWAP               279C DUP
    2878 2DUP                 2810 OVER                2838 ROT
     394 2ROT                  26C PICK                 2D4 ROLL
     288 -ROLL                2798 DROP                2874 2DROP
    286C >R                   282C R>                  27A4 =
    2800 NOT                  289C 0=                   3B4 D0=
     124 0>                   28A0 0<                  2894 U<
    27D0 <                    2784 DU<                  350 D<
```

```
 3C4 D=          27BC >          2748 AND
280C OR          2898 XOR        1418 IF
1438 THEN        1450 ELSE       1484 BEGIN
1504 UNTIL       14DC REPEAT     14BC WHILE
1498 AGAIN       1528 END        1334 DO
13C8 LOOP        13F0 +LOOP      27C8 K
27C4 J           27C0 I          2828 R@
1358 LEAVE       263C EXIT       27CC KEY
27A0 EMIT         A24 ?TERMINAL   31C S->D
2740 ABS          3EC DABS       27EC MIN
 408 DMIN        27E8 MAX         430 DMAX
 5F4 SPACES       248 DEPTH       5A8 CR
 624 TYPE        2774 COUNT      1060 -TRAILING
2808 1+          2880 2+         2804 1-
287C 2-          2884 2/         2888 2*
2778 D+          2780 D-         277C D2/
284C /MOD         148 MOD         480 */MOD
 498 */          2794 UM*         3D4 DNEGATE
27F8 NEGATE      18E8 2CONSTANT  185C CONSTANT
18C0 VARIABLE    2634 SF@        18D4 2VARIABLE
2638 SF!         25F8 FSIN       25FC FTAN
25F4 FCOS        1DC0 F?         1E48 FATAN2
2628 FATAN       25EC F2*        2624 FSQRT
25F0 F2/         1DAC E.         1DD0 F.S
1F38 FNUMBER     1CF0 (F.)       1D98 F.
1D5C (E.)        261C FEXP       25DC F**
25D0 FALOG       25E0 FLOG       25E4 2**X
2620 FLN         1EDC POLY       2630 LOG2
1F1C ODD-POLY    1FE4 FLITERAL   262C FLOOR
25CC FROUND      1C0C PLACES     2614 PI
2618 e           201C FVARIABLE  2608 FLOAT+
25D4 FLOATS      25B0 F!         2050 FCONSTANT
2034 F,          2604 FMIN       25B4 F@
25AC FABS        25E8 F0<        2600 FMAX
25C8 F<          2594 F>D        25D8 F0=
260C FNEGATE     25A4 F/         25A8 S>F
2590 D>F         2598 F+         25A0 F*
259C F-          25C4 FOVER      25C0 FDROP
25BC FSWAP       1BBC FDEPTH     25B8 FDUP
2610 FNIP          80 TOGGLE     2640 FSP
 178 FSP0        2840 RP!        2854 SP!
283C RP@          168 R0         200 UABORT
 1F0 WARNING      AEC MESSAGE    1704 SMUDGE
 F3C DLITERAL     BF4 ?COMP       B70 ERROR
 BD8 ?ERROR       C48 ?CSP        C14 ?EXEC
 C30 ?PAIRS      27B0 @@         C80 ?STACK
  E4 @!          2770 CMOVE>     27A8 EXECUTE
2850 SP@         1924 CODE-SUB   276C CMOVE
 A00 ;S          1984 USER       1904 CODE
1944 END-CODE     7B4 D.          7F4 .
 7C8 .R           77C D.R         804 U.
 7E0 U.R          6F0 SIGN        760 #S
 714 #            814 ?           6D0 #>
 6BC <#           220 BL         10B8 EXPECT
11D4 QUERY                       1E0 STATE
```

```
 190 CURRENT          4F8 CONTEXT          1C8 BLK
 180 DP               1A8 FLD              218 DPL
 1C0 >IN              1B0 BASE             170 S0
 198 TIB              1D0 #TIB             1D8 SPAN
 82C C/L              654 PAD              4A8 HERE
 4B8 ALLOT            4E0 ,                4C8 C,
 5E4 SPACE           2824 ?DUP             504 TRAVERSE
 598 LATEST          12A0 COMPILE          A9C [
 A84 ]               670 HEX              688 DECIMAL
199C ;CODE          1800 <BUILDS         1828 DOES>
 CBC ."              2520 .(             27B8 FILL
 228 ERASE           238 BLANK            6A0 HOLD
 9D4 WORD            D60 CONVERT          E6C NUMBER
1564 FIND           174C ID.             1844 CREATE
12BC [COMPILE]       F14 LITERAL          F98 INTERPRET
171C IMMEDIATE      12CC RECURSE         1304 >MARK
12E4 <MARK          1318 >RESOLVE        12F0 <RESOLVE
1954 :CASE          1294 '               1274 [']
 538 LFA             58C >BODY            544 CFA
 554 NFA             570 PFAPTR           824 B/BUF
19F8 AUTOSTART      1594 UNDO            15D4 FORGET
2298 DUMP           253C .S              2370 WORDS
1228 QUIT           1534 ABORT"           AC4 ABORT
19E8 COLD           274C BRANCH          2820 ?BRANCH
391C ATO4           4090 ADCR            4098 ADDR
40A0 AFMR           40A8 SFF_sa          40B0 SFF_GRP_sa
40B8 EFF_sa         40C0 EFF_GRP_sa      40C8 ENDofTable
40D0 LUTerrAd       40D8 LUTerr          40E0 CANTxSR
40E8 CANRxSR        40F0 CANMSR          40F8 C1MOD
4100 C2MOD          4108 C1CMR           4110 C2CMR
4118 C1GSR          4120 C2GSR           4128 C1ICR
4130 C2ICR          4138 C1IER           4140 C2IER
4148 C1BTR          4150 C2BTR           4158 C1EWL
4160 C2EWL          4168 C1SR            4170 C2SR
4178 C1RFS          4180 C2RFS           4188 C1RID
4190 C2RID          4198 C1RDA           41A0 C2RDA
41A8 C1RDB          41B0 C2RDB           41B8 C1TFI1
41C0 C2TFI1         41C8 C1TID1          41D0 C2TID1
41D8 C1TDA1         41E0 C2TDA1          41E8 C1TDB1
41F0 C2TDB1         41F8 C1TFI2          4200 C2TFI2
4208 C1TID2         4210 C2TID2          4218 C1TDA2
4220 C2TDA2         4228 C1TDB2          4230 C2TDB2
4238 C1TFI3         4240 C2TFI3          4248 C1TID3
4250 C2TID3         4258 C1TDA3          4260 C2TDA3
4268 C1TDB3         4270 C2TDB3          4278 EXTINT
4280 EXTMODE        4288 EXTPOLAR        4290 EXTWAKE
4298 I2C_I2ADR      42A0 I2C_I2CONCLR    42A8 I2C_I2CONSET
42B0 I2C_I2DAT      42B8 I2C_I2SCLH      42C0 I2C_I2SCLL
42C8 I2C_I2STAT  40000018 IOCLR          42D0 IOCLR0
42D8 IOCLR1      40000008 IODIR          42E0 IODIR0
42E8 IODIR1      40000000 IOPIN          42F0 IOPIN0
42F8 IOPIN1      40000010 IOSET          4300 IOSET0
4308 IOSET1         4310 MAMCR           4318 MEMMAP
4320 MAMTIM         4328 PCON            4330 PCONP
4338 PINSEL0        4340 PINSEL1         4348 PINSEL2
```

```
4350 PLLCFG              4358 PLLCON              4360 PLLFEED
4368 PLLSTAT             4370 PWM_CCR             4378 PWM_CR0
4380 PWM_CR1             4388 PWM_CR2             4390 PWM_CR3
4398 PWM_EMR             43A0 PWM_IR              43A8 PWM_LER
43B0 PWM_MCR             43B8 PWM_MR0             43C0 PWM_MR1
43C8 PWM_MR2             43D0 PWM_MR3             43D8 PWM_MR4
43E0 PWM_MR5             43E8 PWM_MR6             43F0 PWM_PC
43F8 PWM_PCR             4400 PWM_PR              4408 PWM_TC
4410 PWM_TCR             4418 RTC_ALDOM           4420 RTC_ALDOW
4428 RTC_ALDOY           4430 RTC_ALHOUR          4438 RTC_ALMIN
4440 RTC_ALMON           4448 RTC_ALSEC           4450 RTC_ALYEAR
4458 RTC_AMR             4460 RTC_CCR             4468 RTC_CIIR
4470 RTC_CTC             4478 RTC_CTIME0          4480 RTC_CTIME1
4488 RTC_CTIME2          4490 RTC_DOM             4498 RTC_DOW
44A0 RTC_DOY             44A8 RTC_HOUR            44B0 RTC_ILR
44B8 RTC_MIN             44C0 RTC_MONTH           44C8 RTC_PREFRAC
44D0 RTC_PREINT          44D8 RTC_SEC             44E0 RTC_YEAR
44E8 SPI0_SPCCR          44F0 SPI0_SPCR           44F8 SPI0_SPDR
4500 SPI0_SPINT          4508 SPI0_SPSR           4510 SPI0_SPTCR
4518 SPI0_SPTOR          4520 SPI0_SPTSR          4528 SPI1_SPCCR
4530 SPI1_SPCR           4538 SPI1_SPDR           4540 SPI1_SPINT
4548 SPI1_SPSR           4550 SPI1_SPTCR          4558 SPI1_SPTOR
4560 SPI1_SPTSR          4568 SPI_SPCCR           4570 SPI_SPCR
4578 SPI_SPDR            4580 SPI_SPINT           4588 SPI_SPSR
4590 SPI_SPTCR           4598 SPI_SPTOR           45A0 SPI_SPTSR
45A8 TIMER0_CCR          45B0 TIMER0_CR0          45B8 TIMER0_CR1
45C0 TIMER0_CR2          45C8 TIMER0_CR3          45D0 TIMER0_EMR
45D8 TIMER0_IR           45E0 TIMER0_MCR          45E8 TIMER0_MR0
45F0 TIMER0_MR1          45F8 TIMER0_MR2          4600 TIMER0_MR3
4608 TIMER0_PC           4610 TIMER0_PR           4618 TIMER0_TC
4620 TIMER0_TCR          4630 TIMER1_CR0          4638 TIMER1_CR1
4640 TIMER1_CR2          4648 TIMER1_CR3          4650 TIMER1_EMR
4658 TIMER1_IR           4660 TIMER1_MCR          4668 TIMER1_MR0
4670 TIMER1_MR1          4678 TIMER1_MR2          4680 TIMER1_MR3
4688 TIMER1_PC           4690 TIMER1_PR           4698 TIMER1_TC
46A0 TIMER1_TCR          46A8 UART0_DLL           46B0 UART0_DLM
46B8 UART0_FCR           46C0 UART0_IER           46C8 UART0_IIR
46D0 UART0_LCR           46D8 UART0_LSR           46E0 UART0_MCR
46E8 UART0_MSR           46F0 UART0_RBR           46F8 UART0_SCR
4700 UART0_THR           4708 UART1_DLL           4710 UART1_DLM
4718 UART1_FCR           4720 UART1_IER           4728 UART1_IIR
4730 UART1_LCR           4738 UART1_LSR           4740 UART1_MCR
4748 UART1_MSR           4750 UART1_RBR           4758 UART1_SCR
4760 UART1_THR           4768 VICDefVectAddr      4770 VICFIQStatus
4778 VICIntEnable        4780 VICIntEnClr         4788 VICIntSelect
4790 VICIRQStatus        4798 VICProtection       47A0 VICRawIntr
47A8 VICSoftInt          47B0 VICSoftIntClr       47B8 VICVectAddr
47C0 VICVectAddr0        47C8 VICVectAddr1        47D0 VICVectAddr10
47D8 VICVectAddr11       47E0 VICVectAddr12       47E8 VICVectAddr13
47F0 VICVectAddr14       47F8 VICVectAddr15       4800 VICVectAddr2
4808 VICVectAddr3        4810 VICVectAddr4        4818 VICVectAddr5
4820 VICVectAddr6        4828 VICVectAddr7        4830 VICVectAddr8
4838 VICVectAddr9        4840 VICVectCntl0        4848 VICVectCntl1
4850 VICVectCntl10       4858 VICVectCntl11       4860 VICVectCntl12
4868 VICVectCntl13       4870 VICVectCntl14       4878 VICVectCntl15
```

```
    4880 VICVectCntl2          4888 VICVectCntl3          4890 VICVectCntl4
    4898 VICVectCntl5          48A0 VICVectCntl6          48A8 VICVectCntl7
    48B0 VICVectCntl8          48B8 VICVectCntl9          48C0 VPBDIV
    48C8 WDFEED                48D0 WDMOD                 48D8 WDTC
    48E0 WDTV                  275C CELL+
    6BF0 END-MACHINE-CHAIN     6BDC MACHINE-CHAIN         6DBC .MACHINES
    6BCC PERIOD                6E28 ISOMAX-START          6C34 NO-MACHINES
    6C48 ALL-MACHINES          6CDC UNINSTALL             6D28 INSTALL
    6C08 MACHINE-LIST          6B88 SCHEDULE-RUNS         6B50 CYCLES
    6B48 EVERY                 6AC4 END-PROC              6A9C PROC
    6AEC AS-TAG                6A6C FOR-INPUT             6A10 FOR-OUTPUT
    6940 WITH-VALUE            6938 SET-MASK              6930 CLR-MASK
    6928 XOR-MASK              6920 AND-MASK              6918 DATA-MASK
    6910 TEST-MASK             6908 AT-ADDR               68C4 IS-STATE?
    6898 SET-STATE             6834 IN-EE                 6828 TO-HAPPEN
    6810 NEXT-TIME             67F8 THIS-TIME             66C4 THEN-STATE
    6698 CAUSES                662C CONDITION             6624 IN-STATE
    6548 ON-MACHINE            6568 APPEND-STATE          6500 MACHINE
    64E0 CURSTATE              64A0 ALLOC                 6498 RAM
    68F4 DEFINE                6398 \                     48F4 SCRUB
    72E8 GET-TIME              72E0 MICROSECONDS          72DC MILLISECONDS
    72E4 SECONDS               7330 MINUTES               72F0 SET-TIMEOUT
    7334 RETIMEOUT             72EC START-TIMEOUT         72D8 START-TIMER
    63B0 TIMEOUT               72F4 TIMEOUT?              7300 0Q
    7320 POP                   7318 PULL                  731C PUSH
    7308 P                     7304 Q                     730C Q?
    7314 QLEFT                 7310 QSIZE                 6408 QUEUE
    63D4 RESIZEQ               7328 ROTATEQ               7324 STUFF
    732C TRANSFERQ             72FC DINT                  72F8 EINT
    7174 S                     48E8 skips                 6390 emitv
    6370 pasm                  6380 pscirxq               6388 pscitxq
    6378 stackerror            3E88 FLWORD           40000020 REDLED
40000028 YELLED           40000030 GRNLED               4900 ON?
    4904 OFF?                  48F8 ON                    48FC OFF
    3934 IAP                   3B3C FLERASE               3C4C FLMOVE
    3DB8 FL!                   3938 FDP                   3924 FUZZIFY
    392C EVALUATE-RULES        3928 DEFUZZIFY             273C SRAND
    2738 RANDOM                2728 RAND_MAX              1554 FORTH-83
OK
```

6. REGISTERS

The ARM micros have quite a few memory mapped registers to control the peripherals. A detailed description of the bits and effects on the peripherals is beyond the scope of this manual so for more information, seek out the user manuals for the particular micro. IsoMax™ carries a superset of the registers even though not all registers are present. The 2106 will have an extra UART while the 2129 will have A/D, CAN bus and an external memory controller. Accessing a register that does not exist on a micro will cause an exception. These registers are grouped by peripheral and act like constants. They can be used with the memory operators @, !, C@ and C! to alter or read the contents just like normal memory.

6.1 A/D Converter

ADCR	ADDR

6.2 Can Bus

AFMR	SFF_sa	SFF_GRP_sa	EFF_sa
EFF_GRP_sa	ENDofTable	LUTerrAd	LUTerr
CANTxSR	CANRxSR	CANMSR	
C1MOD	C2MOD	C1CMR	C2CMR
C1GSR	C2GSR	C1ICR	C2ICR
C1IER	C2IER	C1BTR	C2BTR
C1EWL	C2EWL	C1SR	C2SR
C1RFS	C2RFS	C1RID	C2RID
C1RDA	C2RDA	C1RDB	C2RDB
C1TFI1	C2TFI1	C1TID1	C2TID1
C1TDA1	C2TDA1	C1TDB1	C2TDB1
C1TFI2	C2TFI2	C1TID2	C2TID2
C1TDA2	C2TDA2	C1TDB2	C2TDB2
C1TFI3	C2TFI3	C1TID3	C2TID3
C1TDA3	C2TDA3	C1TDB3	C2TDB3

6.3 External Interrupts

EXTINT	EXTMODE	EXTPOLAR	EXTWAKE

6.4 I^{2}C Interface

I2C_I2ADR	I2C_I2CONCLR	I2C_I2CONSET	I2C_I2DAT
I2C_I2SCLH	I2C_I2SCLL	I2C_I2STAT	

6.5 General Purpose Input/Output

IOCLR	IOCLR0	IOCLR1	IODIR
IODIR0	IODIR1	IOPIN	IOPIN0
IOPIN1	IOSET	IOSET0	IOSET1

6.6 Memory Accelerator Module

MAMCR	MEMMAP	MAMTIM

6.7 Power Control

PCON	PCONP

6.8 Pin Connect Block

PINSEL0	PINSEL1	PINSEL2

6.9 Phase Locked Loop

PLLCFG	PLLCON	PLLFEED	PLLSTAT

6.10 Pulse Width Modulation

PWM_CCR	PWM_CR0	PWM_CR1	PWM_CR2
PWM_CR3	PWM_EMR	PWM_IR	PWM_LER
PWM_MCR	PWM_MR0	PWM_MR1	PWM_MR2
PWM_MR3	PWM_MR4	PWM_MR5	PWM_MR6
PWM_PC	PWM_PCR	PWM_PR	PWM_TC
PWM_TCR			

6.11 Real Time Clock

RTC_ALDOM	RTC_ALDOW	RTC_ALDOY	RTC_ALHOUR
RTC_ALMIN	RTC_ALMON	RTC_ALSEC	RTC_ALYEAR
RTC_AMR	RTC_CCR	RTC_CIIR	RTC_CTC
RTC_CTIME0	RTC_CTIME1	RTC_CTIME2	RTC_DOM
RTC_DOW	RTC_DOY	RTC_HOUR	RTC_ILR
RTC_MIN	RTC_MONTH	RTC_PREFRAC	RTC_PREINT
RTC_SEC	RTC_YEAR		

6.12 Serial Peripheral Interface

SPI0_SPCCR	SPI0_SPCR	SPI0_SPDR	SPI0_SPINT
SPI0_SPSR	SPI0_SPTCR	SPI0_SPTOR	SPI0_SPTSR
SPI1_SPCCR	SPI1_SPCR	SPI1_SPDR	SPI1_SPINT
SPI1_SPSR	SPI1_SPTCR	SPI1_SPTOR	SPI1_SPTSR
SPI_SPCCR	SPI_SPCR	SPI_SPDR	SPI_SPINT
SPI_SPSR	SPI_SPTCR	SPI_SPTOR	SPI_SPTSR

6.13 Timers

TIMER0_CCR	TIMER0_CR0	TIMER0_CR1	TIMER0_CR2
TIMER0_CR3	TIMER0_EMR	TIMER0_IR	TIMER0_MCR
TIMER0_MR0	TIMER0_MR1	TIMER0_MR2	TIMER0_MR3
TIMER0_PC	TIMER0_PR	TIMER0_TC	TIMER0_TCR
TIMER1_CCR	TIMER1_CR0	TIMER1_CR1	TIMER1_CR2
TIMER1_CR3	TIMER1_EMR	TIMER1_IR	TIMER1_MCR
TIMER1_MR0	TIMER1_MR1	TIMER1_MR2	TIMER1_MR3
TIMER1_PC	TIMER1_PR	TIMER1_TC	TIMER1_TCR

6.14 Universal Asynchronous Receivers Transmitters

UART0_DLL	UART0_DLM	UART0_FCR	UART0_IER
UART0_IIR	UART0_LCR	UART0_LSR	UART0_MCR
UART0_MSR	UART0_RBR	UART0_SCR	UART0_THR
UART1_DLL	UART1_DLM	UART1_FCR	UART1_IER
UART1_IIR	UART1_LCR	UART1_LSR	UART1_MCR
UART1_MSR	UART1_RBR	UART1_SCR	UART1_THR

6.15 Vectored Interrupt Controller

VICDefVectAddr	VICFIQStatus	VICIntEnable	VICIntEnClr
VICIntSelect	VICIRQStatus	VICProtection	VICRawIntr
VICSoftInt	VICSoftIntClr	VICVectAddr	VICVectAddr0
VICVectAddr1	VICVectAddr10	VICVectAddr11	VICVectAddr12
VICVectAddr13	VICVectAddr14	VICVectAddr15	VICVectAddr2
VICVectAddr3	VICVectAddr4	VICVectAddr5	VICVectAddr6
VICVectAddr7	VICVectAddr8	VICVectAddr9	VICVectCntl0
VICVectCntl1	VICVectCntl10	VICVectCntl11	VICVectCntl12
VICVectCntl13	VICVectCntl14	VICVectCntl15	VICVectCntl2
VICVectCntl3	VICVectCntl4	VICVectCntl5	VICVectCntl6
VICVectCntl7	VICVectCntl8	VICVectCntl9	

6.16 VPB Divider

VPBDIV

6.17 Watchdog

WDFEED WDMOD WDTC WDTV

7. MEMORY MAP

Great care has been taken to make as much RAM and ROM available for programming. On the LPC2106, there is 32K of flash and 63K of RAM available for user programs. On the LPC2129, it is 160K of flash and 14K of RAM. IsoMax™ occupies the first 32K of the first 64K section and 56K of the second 64K section. The boot block on the LPC2106 is in the upper 8K of the second 64K section and in the upper 8K of the 4th 64K section on the LPC2129.

LPC2106:

0x00000-0x07FFF - 32K flash used by IsoMax™

0x08000-0x0FFFF - 32K flash in 8K sectors for user

0x10000-0x1DFFF - 56K flash used by MaxForth

0x1E000-0x1FFFF - 8K flash used by bootloader

0x40000000-0x40000887 - 2K system RAM

0x40000888-0x4000FFFF - 62K RAM user space and system stack

LPC2129:

0x00000-0x07FFF - 32K flash used by IsoMax™

0x08000-0x0FFFF - 32K flash in 8K sectors for user

0x10000-0x1DFFF - 56K flash used by MaxForth

0x1E000-0x1FFFF - 8K flash unused

0x1E000-0x3DFFF - 120K flash for user in 64K and 8K sectors

0x3E000-0x3FFFF - 8K flash used by bootloader

0x40000000-0x40000887 - 2K system RAM

0x40000888-0x40003FFF - 14K RAM user space and system stack

7.1 Flash

Efficient algorithms for FL!, FLERASE and FLWORD have been carefully crafted to minimize the amount of flash operations to achieve the best response times. The Flash is constrained to 8K and 64K sized and aligned erase blocks called sectors, and writes of only .5K, 1K, 4K or 8K.

When using quick or boot vectors, be careful not to overwrite code you are using. If the vector points to a word in flash and that word gets overwritten, then each time when you reboot, the overwritten word will be run and you will crash. This can be escaped sometimes with a control-g and reset button pushing but might require using the LPC2000 flash utility to refurbish the flash.

When you execute COLD or power cycle the processor, all of RAM is gone and any links or words are also gone. The FDP is reset to 0x8000. If you have stored words in flash with FLWORD and attempt to do it again without first erasing flash, you might end up with problems. In this case, either set FDP to unused flash or erase the part that has been used with FLERASE.

FLERASE (a \ n --) the 8K or 64K memory sectors containing address a for n bytes, is erased which takes about 400ms. Be careful which sectsor you erase as you can render IsoMax inoperable and it will have to be reloaded. FLERASE first checks to see if the sector is not already blank before deciding to erase.

FLMOVE (s \ d \ n --) copy n bytes from address s to address d in flash memory. n must be 512, 1024, 4096 or 8192 bytes.

FLWORD (--) moves the latest word into Flash where the FDP is at and assumes that the flash is set to FF so no erase is needed. Only the needed sectors are written to cover the word length. If the dictionary is not aligned after say a C, then it is first aligned up to a 4 byte boundary before copying to flash. The smallest section of writing for flash is 16 bytes so after a word is moved to flash, FLWORD aligns to the next 16 byte boundary.

FDP (-- a) flash dictionary pointer used by FLWORD to transfer definitions from RAM to flash. This pointer is reset by COLD.

FL! (n \ a --) store n into flash location a. FL! works on any location in flash but it takes up to 400ms.

IAP (a -- a) This calls the IAP code in the boot block with a being the address of the command and response buffer. The buffer is filled with the command and parameters for the IAP call and in return is filled with the response. This is used by the flash programming words.

8. Hands On

8.1 Setup

A good way to prototype small hardware projects directly with the ARM boards is by adding headers to the development board for J8, J9 and J10. Electronic components can directly plug into the headers and save wire wrapping or soldering. This approach was used for the simple battery monitor example.

8.1.1 Small Flash Writes

While the smallest segment that can be written to flash is 512 bytes, it is possible to only modify 16 of those bytes at a time as long as they are on a 16 byte boundary and are all 0xFF. The process involves a read, modify, write back operation. The 512 bytes that are to be modified are read from the flash into a buffer. The 16 bytes to be modified are copied into the relative block within the buffer. The 512 byte buffer is written back to the flash.

9. Examples

Some of these examples deal with the user flash memory which is divided into regions, one common to both the LPC2106 and the extra LPC2129 regions:

- 32K from 0x8000 to 0xFFFF; erase 8K bytes; write 512 bytes; For smaller writes see “Small Flash Writes” on page 75
- 128K bytes extra on the LPC2129: 8K bytes from 0x1_E000 to 0x1_FFFF (erase 64K bytes, write 512 bytes); 64K bytes from 0x2_0000 to 0x2_FFFF (erase 64K bytes, writes 512 bytes); 56K bytes from 0x3_0000 to 0x3_DFFF (erase 8K bytes, write 512 bytes).

Note that because IsoMax™ is in 0x1_0000 to 0x1_DFFF, and not protected, it is less simple to erase the 8k block from 0x1_E000 to 0x1_FFFF. To do this you need to copy out the IsoMax™ code, erase the 64K block from 0x1_0000, and then copy IsoMax™ back. There are two problems with this:

- Where to put 56K of code temporarily (last flash bank),
- and how to do the whole thing without using any code from that 56K block while its region is erased.

The easiest way is to just download IsoMax™ from the LPC2000 utility. The last 8K block, 0x1_E000 to 0x1_FFFF on the LPC2106 and 0x3_E000 to 0x3_FFFF on the LPC2129, contains the boot loader code and is not eraseable.

9.1 Using the ADC to Measure Battery Life

This example uses the ADC to measure the voltage on a battery periodically so that it can be monitored as it drains with different loads and then the values can be graphed. A SSM is used to do the timing and a ASM is used to do the ADC reading.

9.1.1 Code

```
\ Battery life monitor  Rob Chapman  Sep 29, 04

\ The life of a battery can be monitored by connecting
\ it to a fixed load and the AIN0 (pin0 27) and running
\ the state machines.  The main state machine is a ssm
\ while the a/d reader is an asm.  The main state machine
\ starts up things and then periodically reads the battery
\ voltage by getting a timeout and then writing a read command
\ to the A/D.  The asynchronous state machine fields the A/D
\ interrupt when the reading is finished and queues it.

COLD

HEX
\ Tools
: CELLS  ( n -- n' )  0 CELL+ * ;
: LSHIFT  ( n \ m -- n' )  0 DO 2* LOOP ;

\ data structures
100 MILLISECONDS TIMEOUT bto   \ battery sample period
101 QUEUE sampleq        \ queue of samples to flash can be bulk written
CREATE samplearray  200 ALLOT    \ for formmatting for flash

\ ASM - asynchronous state machine for A/D
MACHINE A/D-READER
  APPEND-STATE READ-AIN0

IN-STATE READ-AIN0
  CONDITION 1 CAUSES
```

```
    ADDR @ FFFF AND  6 0 DO 2/ LOOP  sampleq PUSH
    0 ADCR !   \ shut down a/d
    \ ff VICVectAddr !  \ done by IsoMax
  THEN-STATE READ-AIN0 TO-HAPPEN

\ SSM - main state machine
MACHINE BATTERY-MONITOR
  APPEND-STATE MEASURE
  APPEND-STATE DONE

IN-STATE MEASURE
  CONDITION bto TIMEOUT? CAUSES
      200E01 ADCR !  \ ain0:/14:poweron
     1200E01 ADCR !  \ ain0:/14:poweron:start
    sampleq Q? FF >
    IF  samplearray  80 0
        DO  sampleq PULL  10 LSHIFT  sampleq PULL OR  OVER !  CELL+  LOOP
        DROP  FDP @ @ NOT
        IF  20000 FDP !  ." Flash full. " CR
        ELSE  samplearray FDP @ 200 FLMOVE  200 FDP +!  THEN
    THEN
    bto RETIMEOUT
  THEN-STATE MEASURE TO-HAPPEN

IN-STATE MEASURE
  CONDITION  FDP @ 8200 >  CAUSES  \ finished
    REDLED ON
  THEN-STATE DONE TO-HAPPEN

\ The ASM is plugged into the vectored interrupt controller
READ-AIN0 SET-STATE  ' A/D-READER CFA VICVectAddr0 ! 32 VICVectCntl0 !


\ The SSM just installs and a period is chosen
\ requires a bit set to be enabled and a vector to be initialized
bto START-TIMEOUT       \ start the ball rolling
MEASURE SET-STATE  INSTALL BATTERY-MONITOR \ install main state machine as ssm
10 MILLISECONDS PERIOD  \ 10 times shorter than timeout
40000 VICIntEnable !    \ enable A/D interrupts

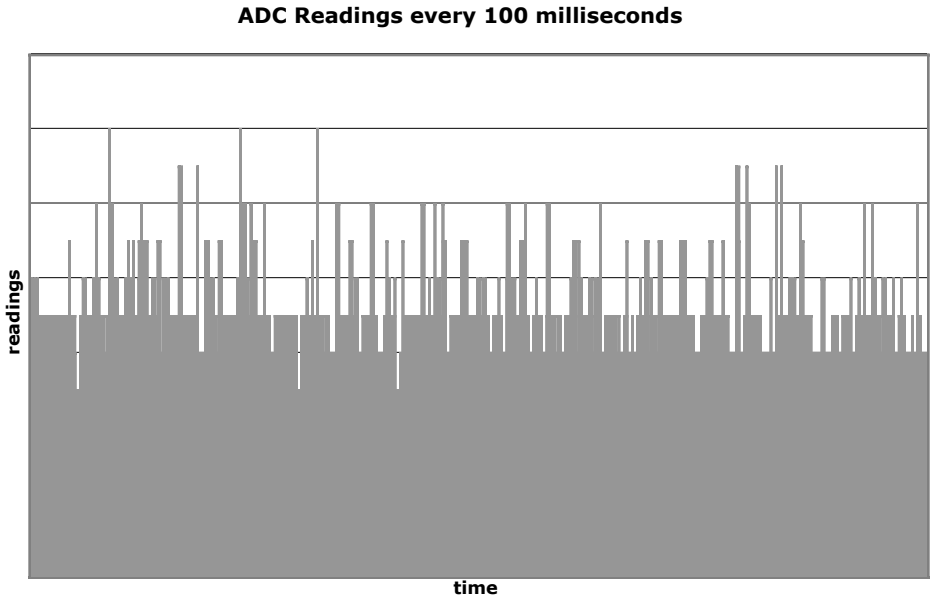
\ Viewing
\ For reading all values stored in flash and queue
: RESET-MONITOR  \ reset the monitor storage
   8000 BEGIN DUP @ NOT WHILE DUP 2000 FLERASE 2000 + REPEAT
   8000 FDP ! ;
: VIEW  \ dump out all the samples and reset
   DECIMAL 8000 BEGIN DUP @ DUP NOT WHILE FFFF AND CR . 2+ REPEAT DROP
   sampleq  BEGIN DUP Q? WHILE DUP PULL CR . REPEAT DROP  RESET-MONITOR CR ;
: SIZE  ( show size of samples )  FDP @ . sampleq Q? . ;
```

9.1.2 Results

The code was downloaded to the PlugAnARM and it runs immediately. A battery was attached and loaded with a low ohmage resistor. Results were viewed at different times in testing out the code.

This graph illustrates the noise on the AIN0 input which should be quite quiet since it is connected to a direct

source of voltage but as can be seen, there are 8 discrete levels read reducing the precision of 10 bits to 7 bits. By combining the readings, bit accuracy can be regained but at a cost of either time or sample rate.



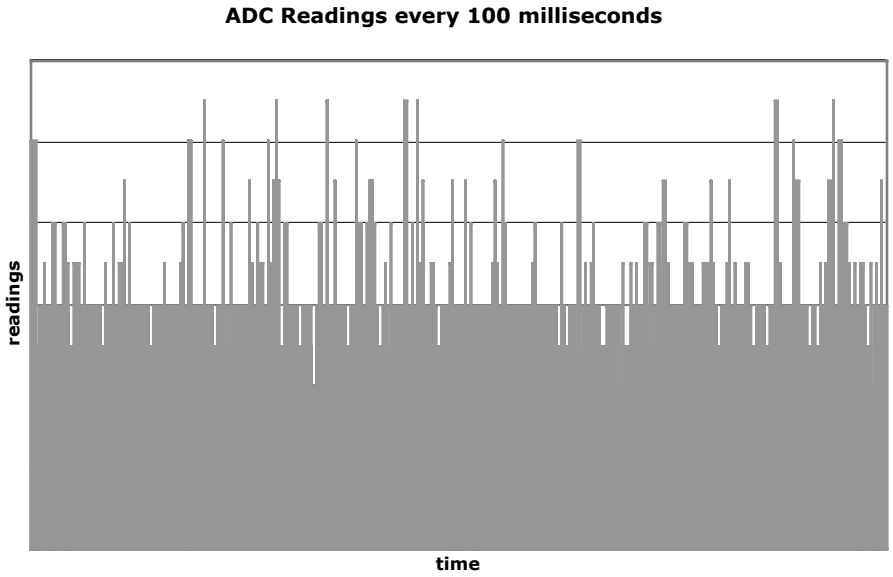
In the ARM LPC user manual it states that if the input pin is set to a digital function instead of the analog function, then the accuracy will be off. But unless the setting has been changed from reset, the default is to have that input be analog so we can rule out that source of noise by simply checking the value in the PINSEL1:

```
HEX PINSEL1 @ U. 15400000  OK
```

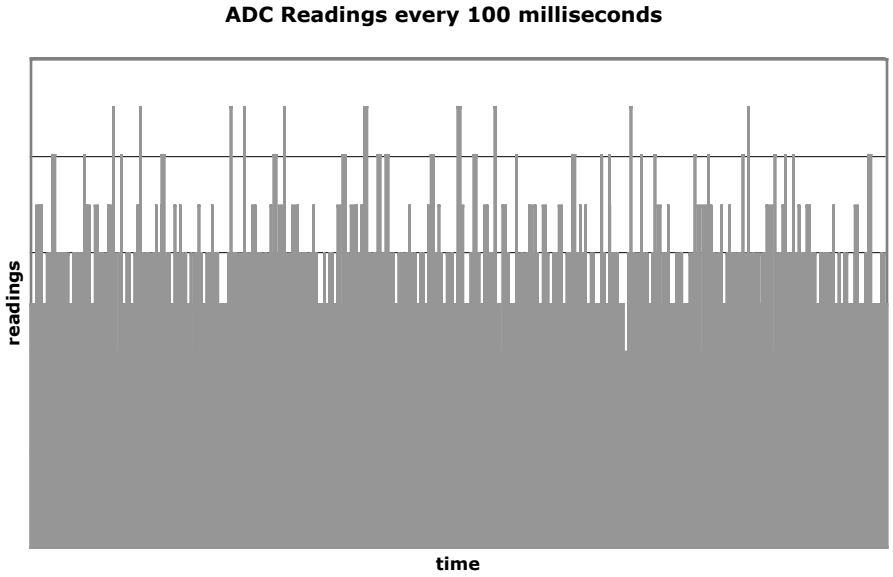
And indeed it is the reset value and the four analog inputs are set as analog. So perhaps the code can be tuned to mitigate any effects of its use. In the code the ADC is turned off after use. This turning on and off of the ADC could be a source of noise. So if we modify the code by commenting out code so that the ADC isn't turned off:

```
\ 0 ADCR !   \ shut down a/d
```

(or turn on and wait) we get pretty much the same thing:



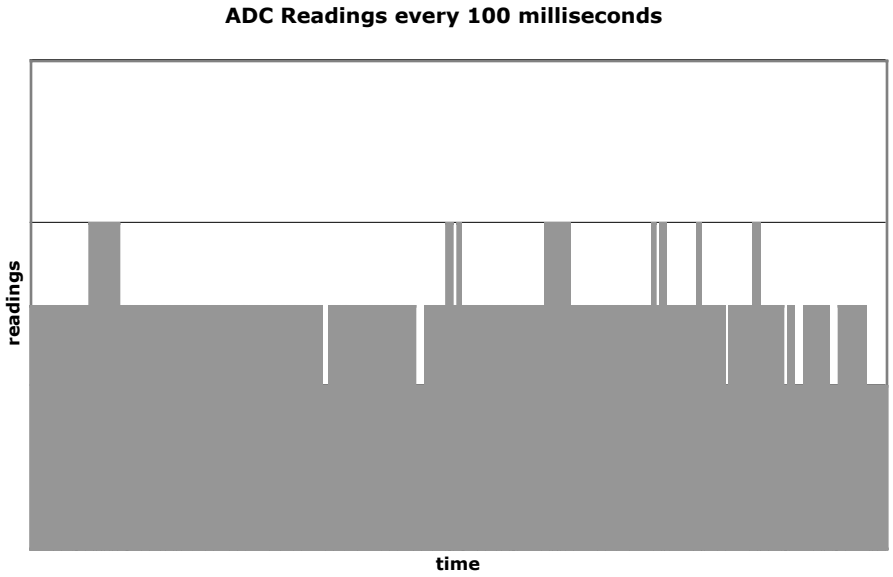
The next thing to try is to change the ADC parameters. If we double the time taken to measure a sample we improve a bit to six discrete levels and if we maximize it, it doesn't get better:



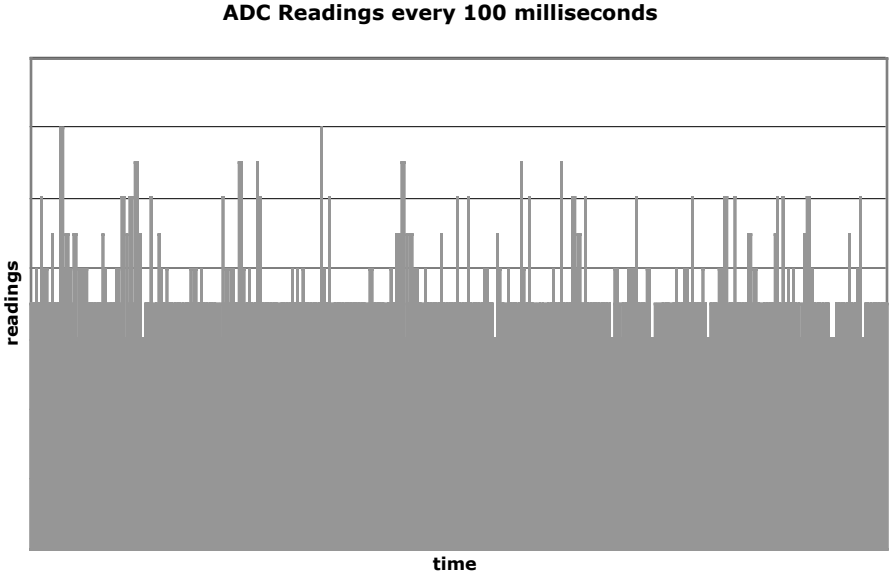
Using a scope to read the input, there is about 10mv of noise which at 10 bits resolution of 3.3V should be within 2 bits or at most, 4 levels.

Next approach is to let ADC continuously sample and then with a SSM only read some of the samples but that produces the same results. So the only way to get back precision is to average. By taking an average of

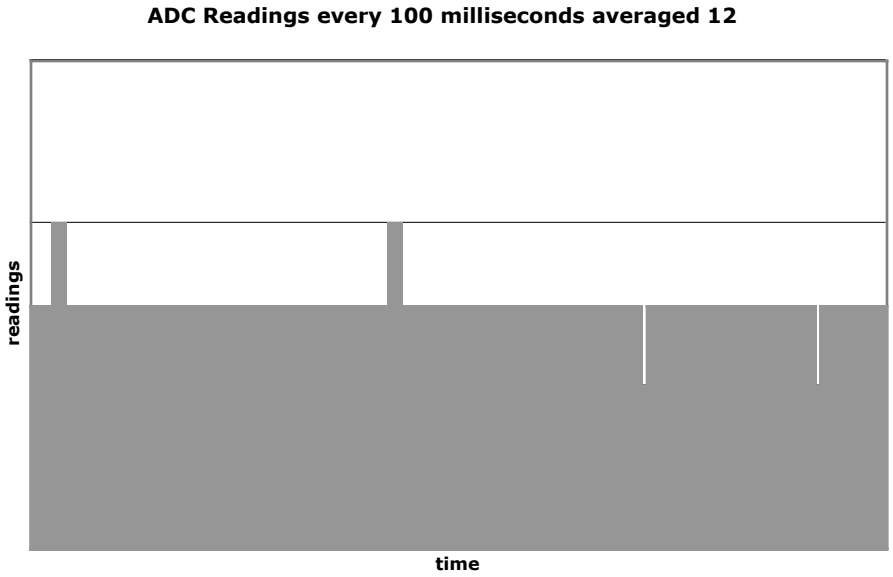
12 samples, the number of discrete levels is reduce to 3:



Using a median of 11 samples we can also reduce the noise. Here is the data without any filtering: This is the result using a similar set of data

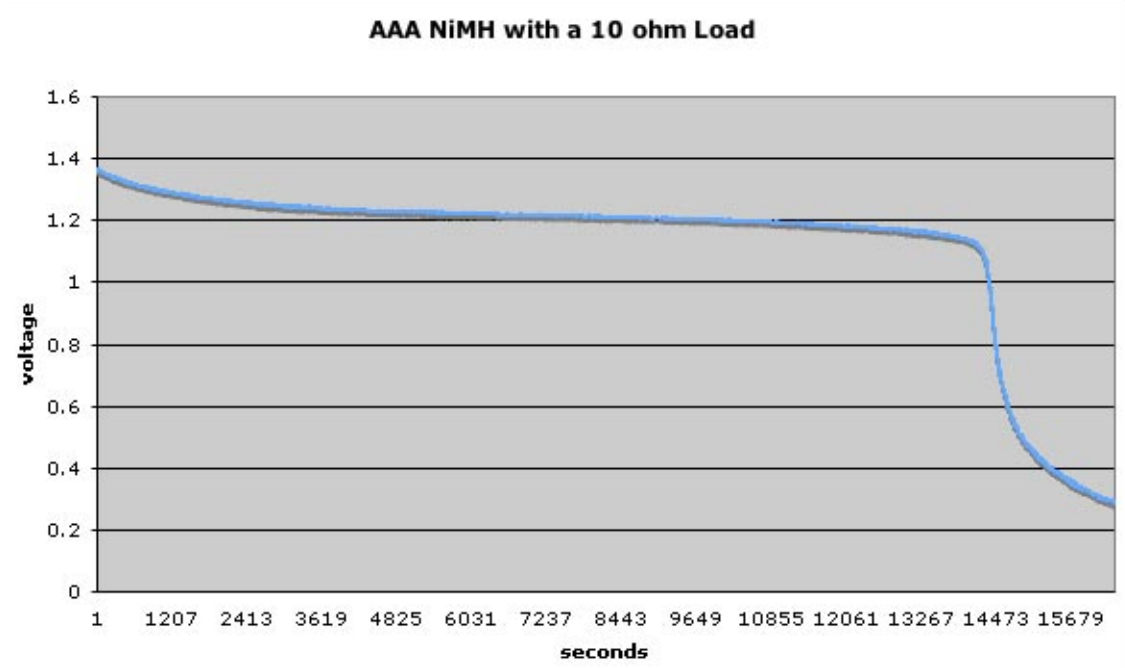


:

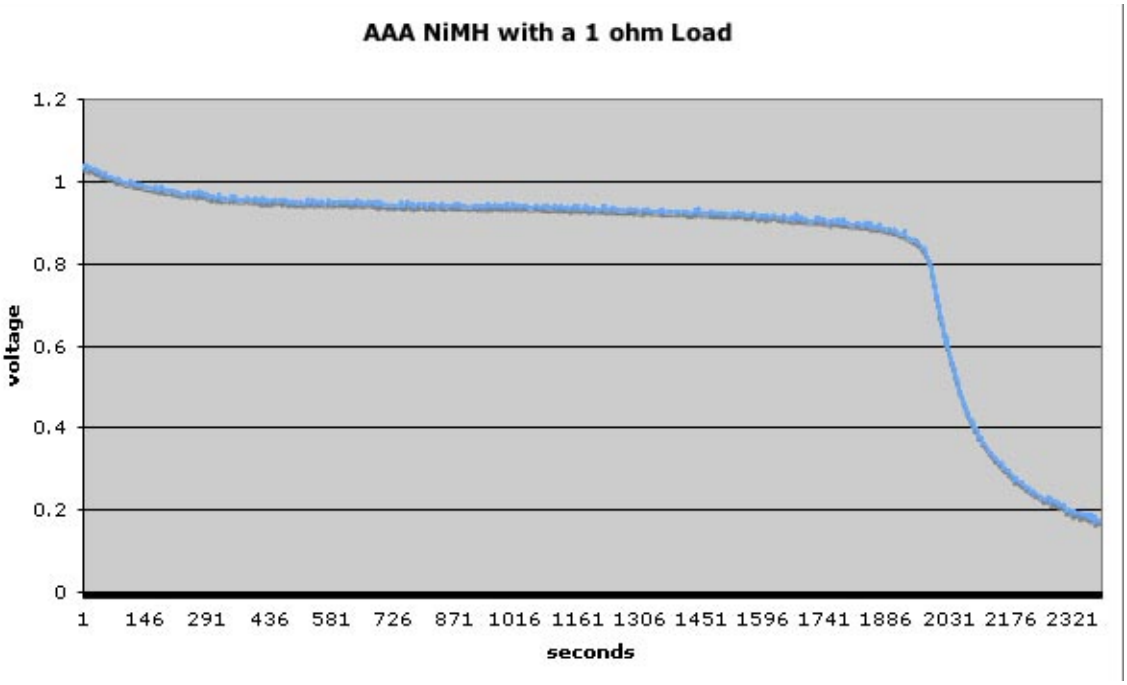


Calculating a median can be a little more expensive than an average since sorting is involved.

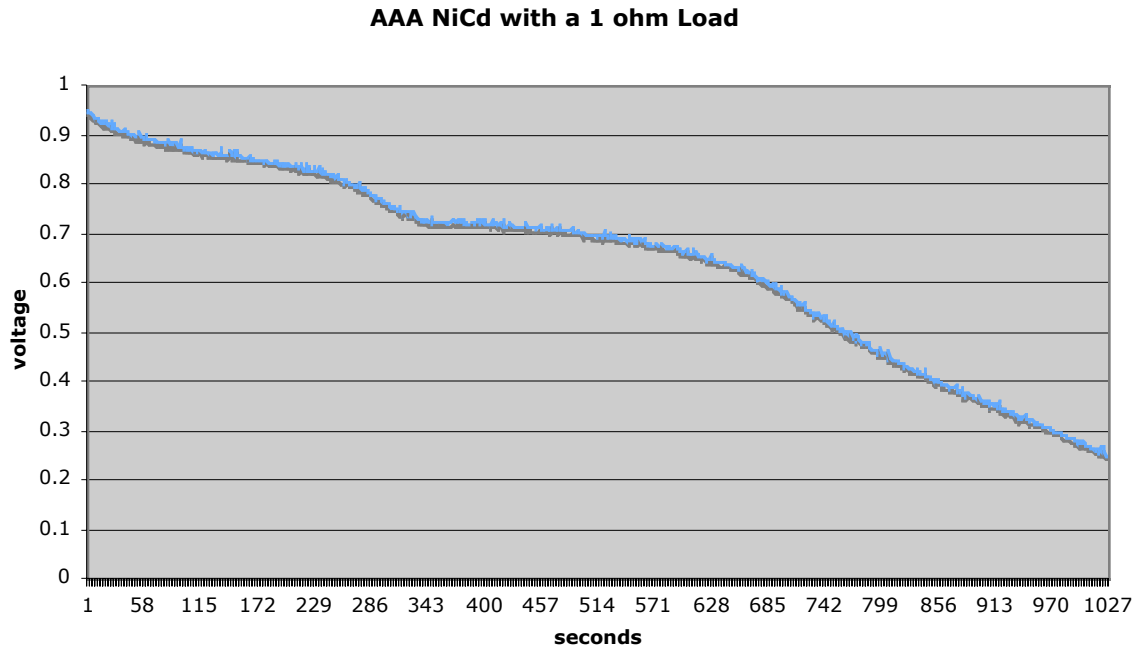
Finally if we accumulate the readings for an AAA NiMh battery with a 10 ohm load we can see the flat voltage mostly out to about 1.1V and then it knees and drops off sharply close to 4 hours:



And with a 1 ohm load we hit the knee at about 33 minutes:



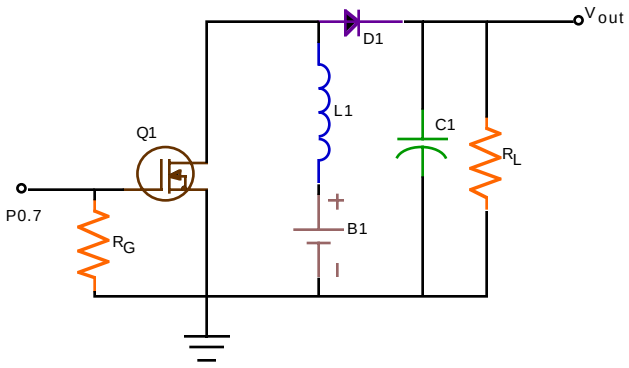
And if we do the same with a NiCD battery, we see that the life is quite a bit shorter or about 10 minutes:



9.2 Using PWM to Generate High Voltage

In this example, the circuit is set up on a bread board and then ground and P0.7 are wired back into the

headers on the ARM Interface Board.



In this example we use a few discrete components to build a voltage magnifier by turning a coil into a temporary current source. In the above circuit, PWM output 2 is used to drive the gate of the Q1 to short out L1 and B1 generating a current in L1. When Q1 shuts off, the current in L1 continues to flow and enough voltage develops across the coil to forward bias the diode D1 and allow the current to charge C1. The load resistor R_L is used to simulate a load. The output voltage is across C1.

9.2.1 Code

```
\ PWM for Voltage Magnification  Rob Chapman  Oct 1, 04

COLD

\ Using a simple circuit, a single battery can be used to generate a much
\ higher voltage.
DECIMAL
VARIABLE period  60000 period !  \ 1000 Hz from 60Mhz clock
VARIABLE onfor   30000 onfor !   \ 50% duty

\ Tools
: `  ( -- n )  BL WORD 1+ C@  [COMPILE] LITERAL ; IMMEDIATE  \ get ASCII code

HEX

: INIT-PWM  ( -- )
   PINSEL0 @  8000 OR  PINSEL0 !   \ enable PWM to control P0.07
   400 PWM_PCR !                   \ enable pwm2 output
   period @ PWM_MR0 !              \ set the period with match 0
   onfor @ PWM_MR2 !               \ set the on for time
   B PWM_TCR !                     \ reset and enable PWM; next 3 grouped
   2 PWM_MCR !                     \ reset counter on match on 0
   9 PWM_TCR ! ;                   \ get the party started

DECIMAL
: DUTY  ( -- f )  onfor @ S>F  period @ S>F  F/ ;

: STATUS  ( -- )  CR  60E6  period @ S>F F/  F>D D. ." Hz  "
   onfor @ S>F  60E-3 F/  F>D D. ." ns  "  DUTY  100E F* F.  ." %" ;

: SET-PWM  period @ PWM_MR0 !  onfor @ PWM_MR2 !  5 PWM_LER ! ; \ alter pwm
```

```
: ADJUST  ( -- )  CR
   ." [ ] -+10% duty; { } -+10% period; S status; Q to quit "
   BEGIN  KEY
     DUP ` Q = IF  0=  ELSE  \ quit by setting key to zero
     DUP ` S = IF  STATUS  ELSE  \ show status
     DUP ` [ = IF  onfor @ DUP 50 / 1 MAX -  1 MAX  onfor !  ELSE \ adjusts...
     DUP ` ] = IF  onfor @ DUP 50 / 1 MAX +  period @ MIN  onfor !  ELSE
     DUP ` { = IF  period @ DUP 50 / 1 MAX -  onfor @ MAX  period !  ELSE
     DUP ` } = IF  period @ DUP 50 / 1 MAX +  60000000 MIN  period !
               THEN THEN THEN THEN  SET-PWM  STATUS  THEN THEN
   0= UNTIL ;
```

9.2.2 Discussion

We could run an algorithm based on the analog input to optimize the pwm signal

to get the maximum voltage. It would be like

Enter state and mark onfor

Increase over a period until voltage falls below start point and then use

half of that as the increase.

Decrease over a period until voltage falls below start point and then use

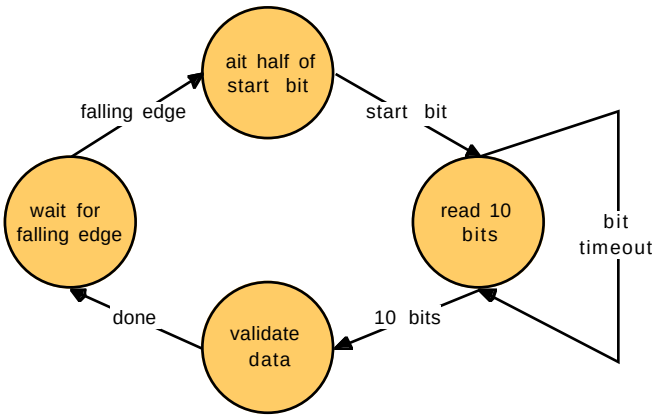
half of tha tas the decrease.

use the periodx2 as the settling time before assessment with a minimum of

1 millisecond

9.3 Software UART for GPS

General purpose pins can be used to emulate a UART in software. This example uses one pin to read the output of a GPS receiver.

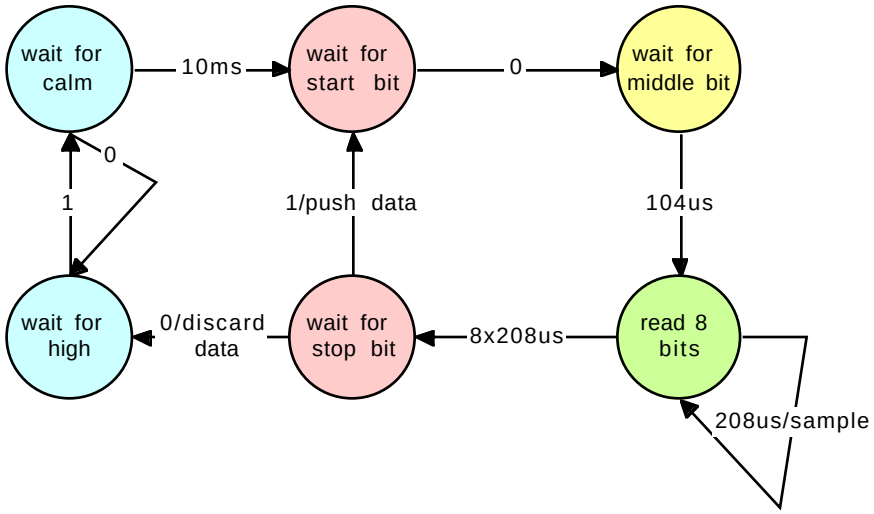


The GPS unit runs on 3.3V and requires as a minimum:

- ground

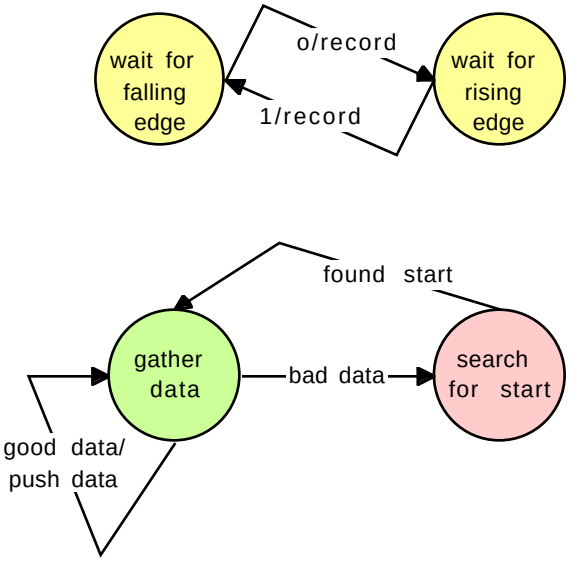
- 3.3V
- 4800 Baud UART receiver

P0.16 is used as the input on the ARM as this goes to external interrupt 0. The output is high unless zero bits like a start bit is being sent. Each bit at 4800 takes 208us and a whole byte with start and stop bits will take 2ms. The data from the GPS is repeated every second and takes 3/4 of a to send the data. So the way to sync up is to wait for the dead space of greater than xms perhaps 10ms or more but less than 200ms. Then wait for EINT0 to go low. This asynchronous trigger will start a periodic state machine which samples the input every 208us centered 100us after the fall for the first start bit. Every 10 bits gathered are then processed as a byte and then the next byte is waited for. If stop bit is low, then resync should be done.



An alternative is to record time of edges and then calculate byte. This is a good method for synching up faster and instream instead of waiting for dead space. This requires rising

and falling edge triggers tho. This requires two state machines. One to record edge



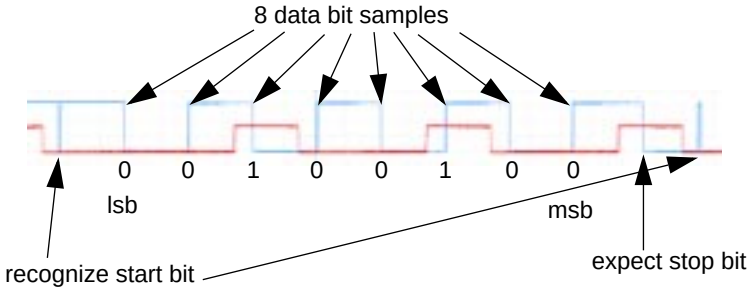
transitions and one to make sense of the data. It also requires a hardware resource which does edge detection. Times are read from a global hardware timer and queued by the first state machine with a level tagged along with the data. The second state machine periodically checks the queue and parses out the timestamps and levels to recreate a data stream which can be sampled in software. To synchronize, a start bit and stop bit pair of transitions are searched for. This is more solid than the first method in that it could sync up within a data byte instead of a cycle which depended on no data.

The first option is just use a timeout and collect bit samples no decoding necessary to get the byte. A variation is to use an ASM and a hardware resource for the timer instead of an SSM and a timeout. Running the SSM clock at a decent rate for 104us timeouts with good accuracy will put a premium on cycle time. For example if the PERIOD is set to anything greater than the 104 us timeout, then the timeout will be too long. If the PERIOD is set to 100 us, then the resolution is poor and if there is jitter, data bits could be read wrong. Since the timeouts are periodic, there won't be slippage in time with respect to the data for such a PERIOD. If PERIOD is set to less than 50 us, it will likely start to run into overrun conditions.

The external interrupt 0 is a level sensitive interrupt. It is enabled with the bit 0 of the register EXTINT. This enables the resource as an interrupt. The interrupt goes to the VIC where it is channel number 14 (0x4000). Since SSM uses timer 0 we will use timer 1 for timing the bit samples. The channel number is 5 (0x20). Making use of the VIC, we can set the state machine up and then for the different states, select the channel number to use as a trigger.

The VIC's flexibility offers us two solutions for tying in the ASM. We could use two

channels, one for each interrupt with both pointing to the ASM and enable/disable the interrupts. Alternatively, one channel could be used and the interrupt number could be swapped in and out as well as managing enables/disables. The former method is chosen.



In reality the state for waiting for the middle of the bit can be dropped since the software is not as fast as hardware, by the time the start bit is detected and then the timer is set up for the data bits, enough time has elapsed that it can be used as a reference point for the next 9 bit samples. In the above figure, the smaller, red waveform is the asynchronous serial data while the taller blue waveform is P0.4 as controlled by TOGS in the states to signal each state. This makes a god way of capture state machine behaviour which depends on timing. When the start bit is found, the P0.4 line is toggled twice to mark for easy capture on an oscilloscope. In some of the states, the timer interrupt is the trigger, in others the external interrupt is used, and in one state both interrupts are active demonstrating both the power of the ARM's vectored interrupt controller and its harnassing by the IsoMax state machine paradigm.

For my software asynchronous receiver, I use P0.16 as EINT0 triggering on a low level and taking that as the start bit. When I get this interrupt my state machine moves into sample mode and uses a second interrupt, timer1, to sample the data bits to make a byte. The problem I ran into is that when P0.16 is set to GPIO (PINSEL1 is xxx0), I can read the incoming serial data fine but when I switch P0.16 to EINT0 function (PINSEL1 is xxx1), then P0.16 always reads as zero even if it is a one. What's the logic in that? So in order to use P0.16 as an interrupt and as a data pin, I have to switch between pin modes using PINSEL1.

LEDS and TOGS are useful debugging aids and can be removed in the final code. LEDS allows one to track state transitions and see the state machine operate in real time to check sequential behaviour. The TOGS is useful for capturing behaviour on a oscilloscope to observe timing.

9.4 Change serial port speed at Bootup

```
: BAUD UART0_LCR C@ DUP [ HEX ] 80 OR UART0_LCR C! OVER UART0_DLL C!
SWAP >< UART0_DLM C! UART0_LCR C! ;  OK
: GB UART0_LCR C@ DUP [ HEX ] 80 OR UART0_LCR C! UART0_DLL C@
UART0_DLM C@ >< OR SWAP UART0_LCR C! ;   OK
```

```
GB . 21  OK
21 BAUD  OK
GB . 21  OK
DECIMAL  OK
GB . 33  OK
60E6 16E F/ FDUP 115200E F/ F. 32.5521  OK
FDUP 9600E F/ F. 390.6250  OK
390 BAUD KEY  OK
: BOOT 390 BAUD ISOMAXBETA ;  OK
' BOOT BOOTUP  OK
```