1. I/O PROGRAMMING

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

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

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

1.1. Bit Output

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

```
pin ON
pin OFF
```

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

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

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

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

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

So, you can turn the red LED on with

```
    REDLED ON
```

and turn it off with

```
    REDLED OFF
```

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

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

So,

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

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

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

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

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

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

1.2. Bit Input

Many of the programmable output pins can be used instead as logic input pins:

```
    PA0 PA1 PA2 PA3 PA4 PA5 PA6 PA7
    PB0 PB1 PB2 PB3 PB4 PB5 PB6 PB7
    PD0 PD1 PD2 PD3 PD4 PD5
    PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7
```

```
TA0 TA1 TA2 TA3
TB0 TB1 TB2 TB3
TC0 TC1
TD0 TD1 TD2 TD3
```

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

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

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

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

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

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

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

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

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

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

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

1.3. Byte Input and Output

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

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

```
PORTA     PORTB
```

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

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

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

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

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

```
HEX 80 PORTA PUTBYTE
```

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

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

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

```
PORTA GETBYTE  INVERT  PORTB PUTBYTE
```

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

1.4. Serial Communications Interface (SCI)

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

```
SCI0     SCI1
```

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

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

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

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

```
HEX 41  SCI0 TX
```

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

```
SCI1 RX  HEX .
```

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

1.4.1. Setting the Baud Rate

When the 'Pod™ is reset, it sets the SCI0 port to operate at 115,200 baud (or 9600 baud for V0.7 or earlier). You can change this to some other value, say 38400, with the command

```
DECIMAL 38400  SCI0 BAUD
```

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

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

```
DECIMAL 9600  SCI1 BAUD
```

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

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

1.4.2. Polling the SCI Status

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

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

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

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

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

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

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

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

1.4.3. Serial Receive Buffering

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

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

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

```
DECIMAL HERE 20 ALLOT CONSTANT BUFFER1
```

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

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

```
    BUFFER1 20 SCI1 RXBUFFER
```

This says that `BUFFER1`, with a length of 20, is to be used as the receive buffer for port `SCI1`. (Note that you use the real buffer length, 20, and not 16.)

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

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

```
    0 0 SCI1 RXBUFFER
```

Returning to unbuffered operation will switch off the SCI receiver interrupt.

1.4.4. Serial Transmit Buffering

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

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

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

```
    DECIMAL HERE 20 ALLOT CONSTANT BUFFER2
```

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

```
    BUFFER2 20 SCI1 TXBUFFER
```

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

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

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

```
0 0 SCI1 TXBUFFER
```

Returning to unbuffered operation will switch off the SCI transmitter interrupt.

1.4.5. Terminal I/O

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

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

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

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

1.4.6. A Serial I/O IsoMax Example

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

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

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

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

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

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

WAITCHAR SET-STATE   INSTALL WATCHSCI1
```

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

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

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

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

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

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

WAITCHAR SET-STATE   INSTALL WATCHSCI1
```

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

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

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

1.5. Serial Peripheral Interface

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

```
SPI0
```

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

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

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

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

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

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

1.5.1. Setting the SPI Parameters

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

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

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

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

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

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

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

1.5.2. Serial Receive Buffering (version 0.6)

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

```
address length port RXBUFFER
```

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

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

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

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

```
    0 0 SPI0 RXBUFFER
```

Returning to unbuffered operation will switch off the SPI receiver interrupt.

1.5.3. Serial Transmit Buffering (version 0.6)

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

```
    address length port TXBUFFER
```

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

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

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

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

```
    0 0 SPI0 TXBUFFER
```

Returning to unbuffered operation will switch off the SCI transmitter interrupt.

1.5.4. An SPI Master-Slave Example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.1. PWM Output

The IsoPod™ and ServoPod™ can generate pulse-width-modulated (PWM) square waves on 26 different output pins. These pins are

```
TA0 TA1 TA2 TA3
TB0 TB1 TB2 TB3
TC0 TC1
TD0 TD1 TD2 TD3[1]
PWMA0 PWMA1 PWMA2 PWMA3 PWMA4 PWMA5
PWMB0 PWMB1 PWMB2 PWMB3 PWMB4 PWMB5
```

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

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

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

[1] Early versions of IsoMax – before version 0.65 – cannot use pin TD3 for PWM operations. On those IsoPods, TD3 may be used for bit I/O. Current IsoPods allow all functions on TD3.

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

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

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

microseconds = N * 0.4

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

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

Also, you're not allowed to set the `PWM-PERIOD` to a value smaller than 4. In other words, the maximum PWM frequency is about 625 kHz.[2] But at that frequency you have only the coarsest control of duty cycle (in 25% steps). To ensure that you have adequate PWM resolution when controlling the duty cycle, `PWM-PERIOD` should be 64 or greater.

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

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

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

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

```
PWMB3 OFF
```

[2] Versions of IsoMax prior to 0.65 did not allow PWM-PERIOD smaller than 256.

2.1.1. Half Speed Operation

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

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

To specify 100 Hz PWM frequency, use the command

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

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

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

2.1.2. Output Polarity

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

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

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

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

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

2.1.3. Complimentary PWM Output

(Version 0.65 and later)

The PWMAx and PWMBx output pins have the ability to operate as complimentary pairs. This means that while the even-numbered pin is high, the odd-numbered pin is low, and vice versa.

`pin COMPLIMENTARY`	puts the pin (and its paired pin) in complimentary mode. This can be used on either pin of the pair, and it affects both; e.g., you can use either `PWMA0 COMPLIMENTARY` or `PWMA1 COMPLIMENTARY` (you don't need to use both, but no harm is done if you do).

PWM output is controlled by the *even* numbered PWM channel, e.g., `PWMA0` in this example. So to output a complimentary signal on `PWMA0` and `PWMA1`, you could write

```
DECIMAL 25000 PWMA0 PWM-PERIOD
PWMA0 COMPLIMENTARY
HEX 4000 PWMA0 PWM-OUT
```

This will output a 25% PWM signal on `PWMA0` **and** a 75% signal (the complimentary signal) on `PWMA1`. You do not need to issue a separate command for `PWMA1`. To return the pins to separate operation, use the command:

`pin INDEPENDENT`	puts the pin (and its paired pin) in independent mode. Again, this can be used on either pin of the pair, and affects both.

Note that using a PWM pin for programmed digital output, with `ON OFF SET` or `TOGGLE`, will automatically set the pin to independent mode.

Why would you use a pair of pins to produce complimentary PWM signals? Some motor drive circuits require complimentary PWM signals, and this can save you from having to add an inverter chip. But the real value of this feature is that it lets you create *nonoverlapping* PWM signals: signals which are never both “on” at the same instant. You do this by specifying a “dead time,” which is the time after one output turns off, before the other output turns on. (During the deadtime, both outputs are off.)

`n pin DEADTIME`	sets the deadtime register. Like `PWM-PERIOD`, this affects **all six** channels of the PWM group. No

special processing is done; the value (0 to 255) is just stored in the deadtime register. This only affects pins in complimentary mode.

A value of 0 given no deadtime, and 255 gives maximum deadtime. The exact amount of deadtime introduced depends on many factors, including the PWM period, and its computation is described in detail in the Motorola DSP56801-7 Users' Manual.

2.2. PWM Input

The IsoPod™ and ServoPod™ can also *measure* pulse-width-modulated (PWM) square waves on the 14 timer pins:

```
TA0 TA1 TA2 TA3
TB0 TB1 TB2 TB3
TC0 TC1
TD0 TD1 TD2 TD3[3]
```

The commands to measure a PWM pulse width are

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

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

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

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

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

microseconds = N * 0.4

[3] Prior to version 0.65, pin TD3 could not be used for PWM operations. On those IsoPods, TD3 may be used only for bit I/O.

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

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

2.2.1. Input Polarity

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

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

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

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

2.2.2. Example

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

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

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

2.3. Analog-to-Digital Conversion

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

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

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

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

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

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

$$V_{in} = 3.3 * N / 32760$$

2.4. Quadrature (Position) Decoders

(Version 0.63 and later)

The IsoPod™ and ServoPod™ can read up to seven position encoders, on 14 pins. Each encoder uses a pair of pins. Two encoders can be connected to dedicated quadrature decoders:

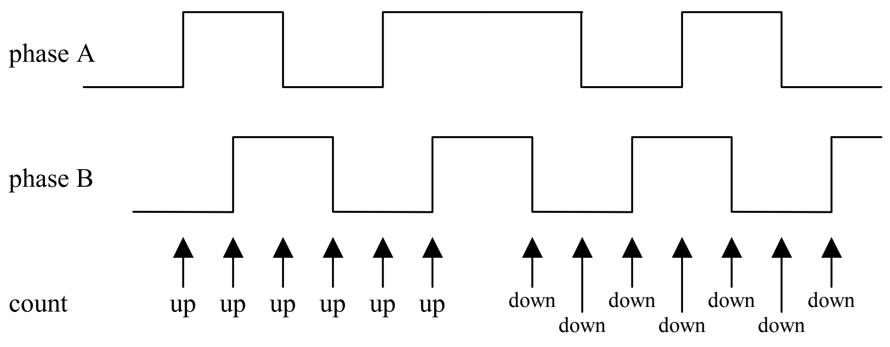
object name	pins used for phase A, B
QUAD0	PHASEA0, PHASEB0
QUAD1	PHASEA1, PHASEB1

The remaining five position encoders can be connected to *pairs* of timer inputs. Timers are always used in pairs, “n” and “n+1”, for this purpose. When you use timer as a position decoder, your code should always refer to the *first* timer of the pair:

object name	pins used for phase A, B
TA2	TA2, TA3 (INDEX0, HOME0)
TB2	TB2, TB3 (INDEX1, HOME1)
TC0	TC0, TC1
TD0	TD0, TD1
TD2	TD2, TD3

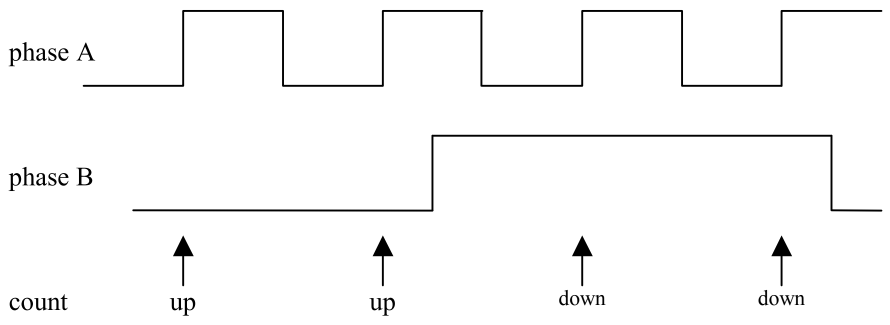
Each encoder uses a pair of input pins, called “phase A” and “phase B.” These can be either quadrature encoded digital inputs (that is, square waves 90° out of phase), or can be separate “count” and “direction” inputs.

If you use quadrature encoding, phase A is the leading phase for a shaft rotating in the positive direction, and phase B is the trailing phase. Every edge on either phase A or phase B will cause a count.



QUADRATURE COUNTING

If you use "count" and "direction," the count input is always the phase A pin; positive edges (low-to-high transitions) are counted. The direction input is always the phase B pin; a logic low means count up, and a high means count down.



COUNT-AND-DIRECTION COUNTING

The commands to control position decoding are:

`pin QUADRATURE`	Sets a *pair* of pins to the quadrature-decoding mode. The acceptable pin names are `QUAD0`, `QUAD1`, `TA2`, `TB2`, `TC0`, `TD0`, and `TD2`.[4]
`pin SIGNED`	Sets a *pair* of pins to the count-and-direction mode. The acceptable pin names are the same as for `QUADRATURE`.

[4] Strictly speaking, you could also specify pins `TA0` or `TB0` for quadrature input. But this is pointless, because these use the same input pins as the dedicated quadrature decoders `QUAD0` and `QUAD1`. If you do specify `TA0` or `TB0`, IsoMax will use the timer registers for counting instead of the dedicated decoder registers. There's no good reason to do this. Use `QUAD0` and `QUAD1` instead.

`pin RESET`	Resets the position counter to zero.
`pin POSITION`	Returns the current position, as a signed double-precision integer on the stack. The position counter is a signed 32-bit counter.

You must specify a decoder mode (`QUADRATURE` or `SIGNED`) before you can use `RESET` or `POSITION`. So, the normal procedure for reading a position encoder is, first, specify the mode, second, reset the position to zero, and then read the position as desired. For example:

```
QUAD0 QUADRATURE
QUAD0 RESET
QUAD0 POSITION D.    ( prints 0 )
     (move shaft)
QUAD0 POSITION D.    ( prints new shaft position )
 . . .

TD0 SIGNED
TD0 RESET
TD0 POSITION D.    ( prints 0 )
     (move shaft)
TD0 POSITION D.    ( prints new shaft position )
```

Note the use of `D.` to print a signed double-precision integer.

<u>Warning:</u> if you use one of the timer pins for output – including the timer pins shared with the `QUAD0` or `QUAD1` decoders – then that will interfere with the use of that pin for decoder input. For example, if you say `TA0 ON`, that will interfere with the `QUAD0` decoder. If you say `TA3 OFF`, that will interfere with the `TA2` decoder (since `TA2` is paired with `TA3` as a decoder).

Also, using one of the timer pins for input will disable the quadrature counter for that pin, so you shouldn't use any of the input functions (like `ON?` and `OFF?`).

Position decoding is done with dedicated counter hardware, so it does not require software intervention to keep an accurate count. The `POSITION` value will always be accurate, regardless of how frequently you read it.

2.4.1. MinPod Encoder Inputs

The MinPod™ has fewer input pins, and can read up to three position encoders:

object name	pins used for phase A, B
QUAD0	PHASEA0, PHASEB0
TA2	TA2, TA3 (INDEX0, HOME0)
TD1	TD1, TD2

This takes advantage of the fact that a "1-2" timer pair can be used for quadrature decoding, as well as a "0-1" or a "2-3" pair. This capability exists on all IsoPod™ and ServoPod™ boards, but it's only important on the MinPod[5].

2.4.2. Index and Home

The "index" and "home" functions of the two dedicated quadrature counters (QUAD0 and QUAD1) are not supported by IsoMax. If you need to use these features, you will need to program the Quadrature Decoder registers explicitly for this. (Refer to Chapter 10 of the Motorola DSP56F801/3/5/7 Users' Manual.)

Note that if you use the INDEX0, HOME0 INDEX1, or HOME1 pins for index or home functions, you cannot use these pins as timer or counter inputs.

2.5. "Software UART" Serial I/O

(Version 0.69 and later)

Sometimes you need more than the two serial channels `SCI0` and `SCI1`. Beginning with IsoMax™ version 0.69, you can use the port E GPIO pins for asynchronous serial input or output:

```
PE2 PE3 PE4 PE5 PE6 PE7
```

To use one of these pins for serial I/O, you must first specify that the pin is to be used for serial data, and whether the pin is an input or an output:

`pin IS-TX`	enables the given pin for serial output
`pin IS-RX`	enables the given pin for serial input

<u>**This step is very important**</u>**.** If you don't enable the pin for serial input or output, the remaining serial functions will malfunction, and may even freeze the 'Pod™. (You won't damage the 'Pod™, but you might have to press RESET or cycle its power.) So remember to use `IS-TX` or `IS-RX`. If you are creating an application which will autostart, be sure to put the required `IS-TX` and `IS-RX` statements in your initialization code.

Of course, a single pin can be either an input or an output, but not both at the same time. To create a full-duplex serial port you will need to use two pins. For example, you might use PE2 for serial input and PE3 for serial output:

```
PE2 IS-RX   PE3 IS-TX
```

[5] Of course, any timer can only take part in *one* pair. You can't use TD0-TD1 *and* TD1-TD2 on an IsoPod, for example. You couldn't connect four encoder signals to three timer pins anyway.

2.5.1. Setting the Baud Rate

You must also specify the baud rate for each pin. There is no default baud rate! Before you can use a pin for serial input or output, you *must* set its baud rate. You do this with the same command you used for the SCI ports:

```
DECIMAL  9600 PE2 BAUD  9600 PE3 BAUD
```

Note that you must specify the baud rate for *each* pin. Since the pins are completely independent, you can specify different baud rates for each. You can specify a baud rate from 300 to 57600 (decimal), but remember the following two restrictions:

RESTRICTION #1. The actual baud rate will be an integer division of a 625 kHz clock, that is, 625000/N. You can specify any baud rate, and the 'Pod™ will give you the nearest baud rate it can attain. But for some rates this will be a poor approximation. You can't get very close to 19200, 38400, or 57600; attempting to use these baud rates will give poor performance. The highest "standard" rate that is accurately supported is 9600 baud (625000/65). Certain baud rates, like 31250 baud, can be exactly reached.

RESTRICTION #2. Because these serial data streams are being sent and received by software, they put a load on the CPU. This load depends on the baud rate. The limit is a total of approximately 60,000 bits per second, all pins combined. So, you can run six pins simultaneously at 9600 baud (6*9600 = 57600).

2.5.2. Sending and Receiving Serial Data

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

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

These work the same as the `TX` and `RX` operations on the SCI port. The data format is fixed as 8 data bits, 1 stop bit, no parity.

Two cautions to keep in mind:

CAUTION #1. If you attempt to transmit on a pin which has not been enabled for transmission with `IS-TX`, `TX` will wait forever for a "transmit ready" condition. It is safest to check with `TX?` before sending a character. (See the next section.)

CAUTION #2. If you attempt to receive on a pin which has not been enabled with `IS-RX`, `RX` will wait forever. It will not see any serial data sent to that pin. It is safest to check with `RX?` before receiving a character. (See the next section.)

2.5.3. Polling the Serial Status

`RX` will *wait* for a character to be received, unless there's already one waiting in the serial data register. This may lead to "Program Counter Capture," where the processor sits in a loop waiting for an external event. Even worse, if you forget to initialize the pin with `IS-RX`, the program will never see the serial input data, and will wait forever.

To avoid this, *poll* the pin before attempting to read it. You do this with

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

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

```
PE2 RX?  IF  PE2 RX  ELSE  -1  THEN
```

will safely check to see if a byte has been received on PA0. If so, it will return the byte; if not, it will return -1. In an IsoMax state machine, you might want the condition `PE2 RX?` to cause a transition to another state, and then read the received character in that new state.

`TX` might also wait, if a previous character hadn't finished transmitting. And if the pin hasn't been enabled with `IS-TX`, `TX` will wait forever! So, even for serial output, it's best to poll the pin.

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

`TX?` will instantly return a true (non-zero) value if the transmitter can accept a character *now*. It will return a zero value if the transmitter is busy, that is, if the transmitter is still sending the last character and can't accept a new one yet. You could use this in procedural code to do something else while waiting:

```
( data ) BEGIN  do-something-else  PE3 TX? UNTIL  PA1 TX
```

but the most effective way to use this is within an IsoMax state machine. When the condition `PE3 TX?` is true, you can cause a transition to a new state that outputs the data.

2.5.4. Serial Buffering

When a GPIO pin is used for serial input, it has a single-character buffer. The first character received is held in a buffer while the second character is being shifted in. As soon as the second character is completely shifted in, it will be transferred to the receive buffer. So, you have only one byte period (ten bit periods) to notice and read the received character.

GPIO pins used for serial output are not buffered. You cannot send another character until the first character has been completely shifted out (including its stop bit). This means that -- unless you sit in a *very* tight loop waiting to do output – you will not achieve the full theoretical data rate for the serial channel. There will always be short delays between characters. Normally this is not problem, but you should be aware of this limitation.

The GPIO pins do *not* have the ability to define larger buffers with `RXBUFFER` and `TXBUFFER`.

If you are expecting a continuous stream of serial data at high baud rates, you should use the `SCI0` or `SCI1` ports. The GPIO pins are better suited to lower baud rates, or to protocols (such as half-duplex or packet transmission) where the arrival of data can be predicted.

2.5.5. Stopping and Starting the Serial Engine

Normally it will not be necessary to stop the software "engine" that does serial I/O. When there is no serial input or output activity on the GPIO pins, the engine will automatically go into a quiescent state and will use no CPU cycles.

For debugging purposes, though, you can shut off the engine with the command

`SSTOP`	turn off the serial I/O processing

You turn the software engine back on with the command

`SSTART`	initialize and activate the serial I/O processor

<u>CAUTION #3.</u> You will need to do this if you do a `COLD` or a `SCRUB`. As a safety feature, `COLD` and `SCRUB` turn off all timer functions, including all IsoMax state machines and the serial engine. So after you do a `COLD`, the serial I/O functions will not work until you either do `SSTART` or reset the processor.

2.6. Indexed Pin I/O (Pin Numbering)

(Version 0.68 and later)

Sometimes you'd like to refer to the input/output pins by number instead of by name. Perhaps you need to write a program that receives a number N from 0 to 7, and must send back the current value of analog input N. Or perhaps you want to treat eight input bits as an array. To enable this kind of programming, IsoMax now supports pin numbering.

To reference a pin by number, you use the phrase

```
    n PIN
```

instead of the pin’s name. “n” need not be explicitly written. It can be fetched from a variable, or from a loop index, or the result of a calculation.

2.6.1. Pin Numbering

Pin #	TiniPod	PlugaPod	MinPod	IsoPod	ServoPod
1	*n/a*	*n/a*	PA0	PA0	PA0
2	*n/a*	*n/a*	PA1	PA1	PA1
3	*n/a*	*n/a*	PA2	PA2	PA2
4	*n/a*	*n/a*	PA3	PA3	PA3
5	*n/a*	*n/a*	PA4	PA4	PA4
6	*n/a*	*n/a*	PA5	PA5	PA5
7	*n/a*	*n/a*	PA6	PA6	PA6
8	*n/a*	*n/a*	PA7	PA7	PA7
9	ISA0/PWMA0	ISA0/PWMA0	PE3	PB0	PB0
10	ISA1/PWMA1	ISA1/PWMA1	PE4	PB1	PB1
11	ISA2/PWMA2	ISA2/PWMA2	PE5	PB2	PB2
12	FAULTA0/ PWMA3	FAULTA0/ PWMA3	PE6	PB3	PB3
13	FAULTA1/ PWMA4	FAULTA1/ PWMA4	PE7	PB4	PB4
14	FAULTA2/ PWMA5	FAULTA2/ PWMA5	TA0/PHASEA0	PB5	PB5
15	TD1	TD1	TA1/PHASEB0	PB6	PB6
16	TD2	TD2	TA2/INDEX0	PB7	PB7
17	TA0	TA0	TA3/HOME0	PE2	PE2
18	TA1	TA1	TD1	PE3	PE3
19	TA2	TA2	TD2	PE4	PE4
20	TA3	TA3	PWMA0	PE5	PE5
21	PE4	PE4	PWMA1	PE6	PE6
22	PE5	PE5	PWMA2	PE7	PE7
23	PE6	PE6	PWMA3	TA0/PHASEA0	TA0/PHASEA0
24	PE7	PE7	PWMA4	TA1/PHASEB0	TA1/PHASEB0
25		*n/a*	PWMA5	TA2/INDEX0	TA2/INDEX0
26		*n/a*	FAULTA0	TA3/HOME0	TA3/HOME0
27		*n/a*	FAULTA1	TB0/PHASEA1	TB0/PHASEA1
28		*n/a*	FAULTA2	TB1/PHASEB1	TB1/PHASEB1
29		*n/a*	FAULTA3	TB2/INDEX1	TB2/INDEX1
30		*n/a*	ISA0	TB3/HOME1	TB3/HOME1
31		*n/a*	ISA1	TC0	TC0
32		*n/a*	ISA2	TC1	TC1
33		ADC1	ADC0	TD0	TD0
34		ADC0	ADC1	TD1	TD1
35		ADC3	ADC2	TD2	TD2
36		ADC2	ADC3	TD3	TD3
37		ADC5	ADC4	PWMA0	PWMA0
38		ADC4	ADC5	PWMA1	PWMA1
39		ADC7	ADC6	PWMA2	PWMA2
40		ADC6	ADC7	PWMA3	PWMA3
41		PA1		PWMA4	PWMA4
42		PA0		PWMA5	PWMA5
43		PA3		PWMB0	PWMB0
44		PA2		PWMB1	PWMB1
45		PA5		PWMB2	PWMB2
46		PA4		PWMB3	PWMB3

Pin #	TiniPod	PlugaPod	MinPod	IsoPod	ServoPod
47		PA7		PWMB4	PWMB4
48		PA6		PWMB5	PWMB5
49				FAULTA0	FAULTA0
50				FAULTA1	FAULTA1
51				FAULTA2	FAULTA2
52				FAULTA3	FAULTA3
53				FAULTB0	FAULTB0
54				FAULTB1	FAULTB1
55				FAULTB2	FAULTB2
56				FAULTB3	FAULTB3
57				ISA0	ISA0
58				ISA1	ISA1
59				ISA2	ISA2
60				ISB0	ISB0
61				ISB1	ISB1
62				ISB2	ISB2
63				ADC0	ADC0
64				ADC1	ADC1
65				ADC2	ADC2
66				ADC3	ADC3
67				ADC4	ADC4
68				ADC5	ADC5
69				ADC6	ADC6
70				ADC7	ADC7
71					ADC8
72					ADC9
73					ADC10
74					ADC11
75					ADC12
76					ADC13
77					ADC14
78					ADC15

For the **TiniPod,** the pin numbers correspond to the numbering of the J1 connector.

For the **PlugaPod,** the first 24 pin numbers correspond to the numbering of the J1 connector. The next 24 pin numbers correspond to the numbering of the J5 connector (offset by 24).

For the **MinPod, IsoPod, and ServoPod,** since they each have several I/O connectors, the pin numbering does not correspond directly to any of them. Instead, the pin numbering is "functional," as shown in the table above.

2.6.2. Supported I/O Functions

Only a limited set of functions can be used with `PIN`.

If you use a function which is not supported on a given pin – such as trying to do output on an input-only pin, or ANALOGIN on a digital pin – the 'Pod™ will take what it considers the most reasonable action. This action will always have the expected stack effect (i.e., it will take something from the stack, or put something on the stack). The 'Pod™ will also do this if you specify an invalid pin number.

The supported functions are:

ON, OFF, TOGGLE …on pins capable of digital output, these will have the expected result. On pins which are "input-only" pins (`FAULTxx`, `ISxx`, `ADCx`), these will do nothing.

SET …on pins capable of digital output, this will have the expected result. On pins which are "input-only," this will simply discard the input argument.

ON?, OFF? …on pins capable of digital input, these will have the expected result. On the analog input pins (`ADCx`), both ON? and OFF? will always return zero.

The "output-only" pins (`PWMxx`) are a special case. On the MinPod, IsoPod, and ServoPod, these will return the value last output to the pin (i.e., they will perform the functions of `?ON` and `?OFF`).

On the TiniPod and PlugaPod, each of these output pins is paired with an input pin, so each pin *does* have an input function. Thus, on the TiniPod,

`9 PIN ON`	is equivalent to	`PWMA0 ON`	but
`9 PIN ON?`	is equivalent to	`ISA0 ON?`	

Note that if you have used such a pin for output, the output will remain enabled, and so `ON?` and `OFF?` will return the current output state.

ANALOGIN …on analog input pins, this will have the expected result. On pins capable of digital input, this will return $FFFF if the input is high, or 0 if the input is low; i.e., the same function as `ON?`. On "output-only" pins, this will act the same as `ON?` (see above).

PWM-PERIOD, PWM-OUT …on pins capable of PWM output (`PWMxx` and `Txx`), these will have the expected result. On all other pins, these will simply discard the input parameters.

For all other I/O operations, you must refer to pins by name. In particular, the serial I/O operations (TX, RX, etc.) cannot be used with PIN. You must name the SCI port or GPIO pin that you wish to use.

2.6.3. Examples

This word, on the IsoPod™, will print the current values of all eight analog inputs:

```
DECIMAL 63 CONSTANT FIRST-ADC
        71 CONSTANT LAST-ADC+1
```

```
: .ANALOGS
    LAST-ADC+1 FIRST-ADC DO   I PIN ANALOGIN .  LOOP
;
```

This word, on the MinPod, will return true if any of the timer inputs is high.

```
DECIMAL 14 CONSTANT FIRST-TIMER
        20 CONSTANT LAST-TIMER+1
: ANY-HIGH? ( -- f )
    0  LAST-TIMER+1 FIRST-TIMER DO   I PIN ON? OR  LOOP
;
```

This word, on the PlugaPod, will set the duty cycle of the six PWM outputs to match the analog values read on the first six analog inputs. (It assumes that PWM-PERIOD has already been set.) This would allow six potentiometers, connected to ADC0-5, to control the speed of six motors, connected to PWMA0-5.

```
DECIMAL 9 CONSTANT FIRST-PWM
        15 CONSTANT LAST-PWM+1
        33 CONSTANT FIRST-ADC
FIRST-ADC FIRST-PWM - CONSTANT ADC-OFFSET
: ADC-TO-PWM
    LAST-PWM+1 FIRST-PWM DO
        I ADC-OFFSET + PIN ANALOGIN  I PIN PWM-OUT
    LOOP
;
```

2.7. *Using Trinaries for I/O*

The input and output "methods" that have been described in this chapter are intended to be easy to use, and to get you up and running quickly. So, most of them will perform any port initialization that is required, even if the port has already been initialized. Trinaries are defined in section 4.9.