

Using the Propeller 2 Monitor

DRAFT 0.8

Doug Dingus, January 2013

Table of Contents

| | |
|--|----|
| Using the Propeller 2 Monitor | 1 |
| Table of Contents..... | 2 |
| Introduction | 3 |
| Propeller 2 ROM / RAM Memory Layout..... | 3 |
| Propeller 2 Boot Sequence..... | 4 |
| <i>Recommended Software Tools</i> | 5 |
| Propeller Serial Terminal..... | 5 |
| PUTTY..... | 5 |
| Prop Terminal | 5 |
| Configure your Serial device. | 6 |
| Connect and Identify COM Port for Prop Plug | 6 |
| Connect Prop Plug to Emulated Propeller 2..... | 8 |
| Configure PUTTY..... | 10 |
| Power Up Quick Test | 10 |
| Using the Monitor Basics..... | 11 |
| Built in help screen..... | 11 |
| Command Groups..... | 11 |
| Basic Task Examples..... | 12 |
| Display Contents of HUB Memory | 12 |
| Modify Contents of HUB Memory | 13 |
| Patterns, Block Moves and Search..... | 14 |
| Write Multiple Values to an Address Range..... | 14 |
| Block Move Linear Chunks of RAM | 14 |
| Perform a Search within Data Range | 15 |
| Plain Text Input..... | 16 |
| Watch Pins, Addresses and Set Various States..... | 16 |
| Watch an Address..... | 16 |
| Configure I/O Pin and Set Pin State | 16 |
| Watch an I/O pin | 17 |
| SET COG DAC Values | 17 |
| Set the Propeller Clock (CLKSET)..... | 18 |
| More Advanced Examples..... | 19 |
| Transfer small PASM program to HUB and run it on a COG | 19 |
| Display COG State Map, Run the Program On Various COGS | 20 |
| Set BYTE, WORD, LONG modes..... | 20 |
| Use Paste to Upload Programs and Data through Monitor..... | 21 |
| Copy Data File into Propeller 2 through Monitor..... | 22 |
| Compute a Checksum..... | 23 |
| Launch Monitor From Within Your Program | 23 |
| Modify Running PASM Program..... | 24 |
| Closing..... | 27 |
| Appendix A Propeller 2 ROM Program Listings | 28 |
| ROM_Booter..... | 28 |
| ROM_SHA256 | 32 |
| ROM_Monitor | 37 |
| Appendix B PASM Program Listings and Object Code | 50 |
| DE2-Counter-To-LED-Blinker | 50 |
| Running Multiple Monitors..... | 50 |
| Start Monitor From HUB RAM Memory..... | 51 |
| Replace_Example.spin..... | 51 |

Introduction

Every Propeller 2 chip comes with a built in system monitor that you can use to have a low level conversation with the chip as needed for your development, testing or experimentation using just about any serial device capable of ASCII communications. No programming tools are required to use the monitor, though they are recommended to generate code to be uploaded or better understand the contents of shared HUB or core COG memory.

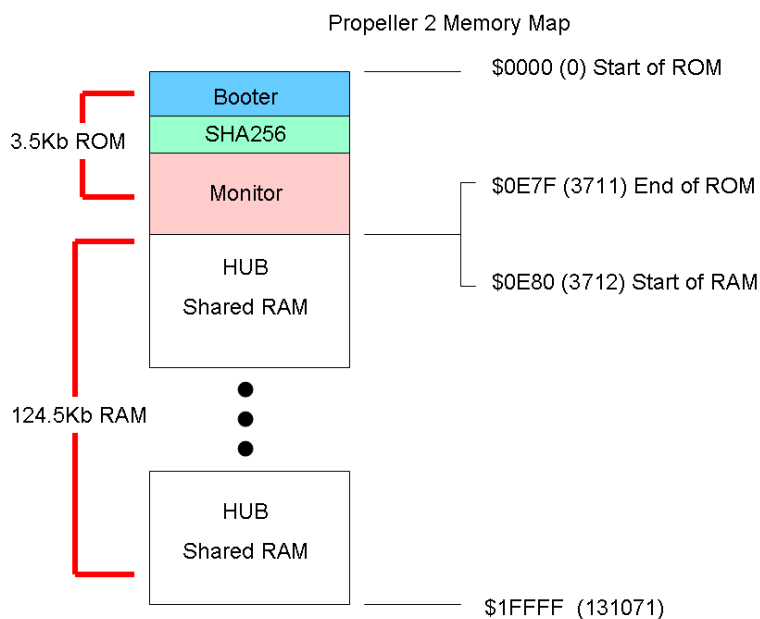
This book contains information on how to use the monitor in a variety of practical and useful ways as well as basic chip information, such as ROM contents, memory layout and other things related to the monitor. This book does not cover PASM programming beyond a simple example or two necessary to demonstrate some advanced monitor functions.

For many users, the monitor can be the very first thing you do to get familiar with the Propeller 2!

Propeller 2 ROM / RAM Memory Layout

The Propeller 2 chip has three core PASM programs in ROM. These are: ROM_booter, ROM_SHA256, and ROM_Monitor. Please see program listings in Appendix A at the end of the document to examine them in detail and read the basic explanation text at the start of each listing.

Both the Monitor and SHA256 ROM routines can be used from within your program. This guide only covers starting the monitor.



Propeller 2 has a 128Kb, 17bit memory address space ranging from \$00000 to \$1FFFF. Both ROM and RAM exist within this address space. Addresses \$000 to \$0E7F contain the built in ROM programs. Writable, shared HUB memory extends from \$E80 all the way through to \$1FFFF, for a total of 3.5Kb ROM and 124.5Kb RAM.

Propeller 2 Boot Sequence

When powered up, the Propeller 2 runs the booter program which controls the boot sequence. There are three possible boot options, in this order:

1. Serial
2. SPI Flash
3. Monitor

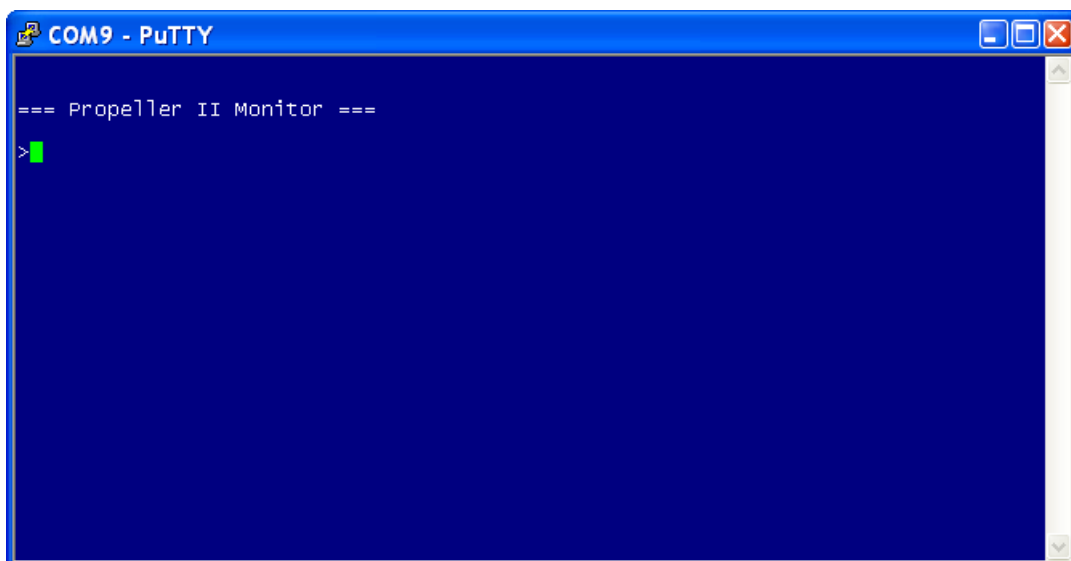
Serial and Flash both require external data be sent to the chip, and optionally pass encryption authentication tests.

If both of those fail, the monitor is activated on pins **90(rx)** and **91(tx)** by default. **The monitor will not activate if the encryption fuses have been set on a particular Propeller 2 chip.** A value of 0 is required to be present for the monitor to operate so that the chip is secure when it's fuse bits are set for that purpose.

Once the monitor is activated, pressing the space bar on your serial capable device allows the auto baud routine to synchronize and enable communications.

This is the default way to enter the monitor. Since it can be initiated from your program, the monitor can be assigned to other I/O pins and started in a variety of ways.

Here is what the monitor looks like after it has been activated and you have pressed the space bar to sync up the auto baud task:

A screenshot of a PuTTY terminal window titled "COM9 - PuTTY". The terminal has a dark blue background and white text. The text displayed is "=== Propeller II Monitor ===" followed by a green cursor and a greater-than sign ">".

```
COM9 - PuTTY
=== Propeller II Monitor ===
>
```

Recommended Software Tools

Basically the only tool you need to interact with the monitor is some kind of serial terminal emulation. Just about any device you can think of which is capable of plain ASCII communication will operate just fine with the Propeller monitor. Your author has used an Apple 2 home computer, and is eager to try a TRS-80 Model 100 portable computer.

The monitor does not output any special ASCII characters, other than the system bell on user input error. Only standard ASCII characters \$20 - \$7E are communicated.

Here is a short list of common terminal software options.

Propeller Serial Terminal

You can obtain the Propeller Serial Terminal via download from Parallax.com here: <http://www.parallax.com/Portals/0/Downloads/sw/propeller/Parallax-Serial-Terminal.zip>

PUTTY

Putty is a very capable, configurable, fast and flexible serial terminal interface. Your author highly recommends PUTTY as a serial terminal. PUTTY can be obtained here: <http://www.putty.org/>

PUTTY will be used for most examples in this book. Feel free to use whatever terminal software or device you prefer.

Prop Terminal

This program was written by Andy Schenk. (Ariba) and is maintained and distributed on the Parallax Support Forums. Here is a thread that has an updated P2 version capable of utilizing the monitor to upload object files to a Propeller 2:

<http://forums.parallax.com/showthread.php?144199-Propeller-II-Emulation-of-the-P2-on-DE0-NANO-amp-DE2-115-FPGA-boards&p=1150859&viewfull=1#post1150859>

And an older forum thread where updates are regularly posted:

<http://forums.parallax.com/showthread.php?94310-Updated-the-PropTerminal>

The Prop Terminal is special in that it renders the terminal session to a graphics window and supports many graphics functions through the serial connection. If you need graphical data verification and do not have video support connected to your Propeller 2 chip yet, this terminal is a great option.

Configure your Serial device.

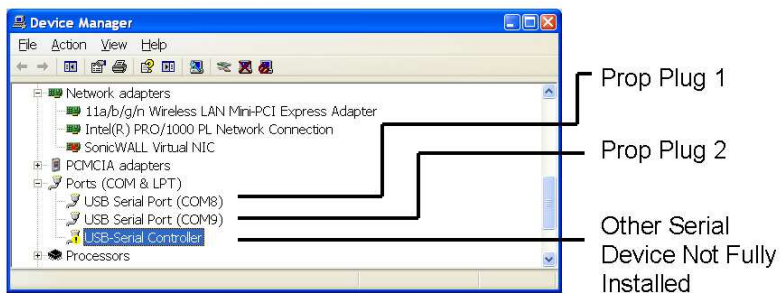
Let's test the Propeller 2 chip and serial terminal software you have chosen. Since the monitor does auto baud, you need only know which COM port your serial device is talking through.

Due to the extremely wide variety of serial options available, this guide will only cover the Prop Plug USB Serial Interface at this time. These instructions also work well for the onboard Serial Interfaces found on many Parallax development boards, though the monitor is specific to Propeller 2.

Connect and Identify COM Port for Prop Plug

Connect your Prop Plug to your PC USB port and verify the device has been recognized and enumerated. You can use the Windows Device Manager for this. Either navigate through the Control Panel, System applets, or input "mmc devmgmt.msc" into your Windows Start Menu, Run dialog, or a DOS command prompt to launch Device Manager, as seen on the following page.

Device Manager Showing Two Prop Plugs Connected to Windows XP System. (Other Windows Operating Systems Similar.)



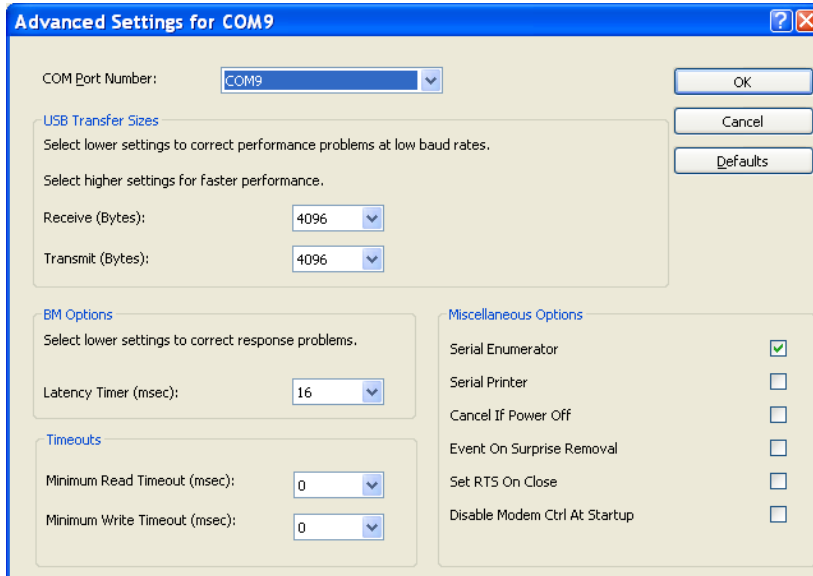
To determine the port when multiple serial devices are present, disconnect, then connect the Prop Plug USB while viewing the Device Manager to see it's COM port assignment appear in the list.

Only one Prop Plug is needed, however two can be advantageous when the monitor is called from a user program. Two are shown in the screen capture above along with another generic USB to Serial device which could be used with the monitor. Note the little yellow or red icon that may appear next to your COM port assignment, as seen on the generic USB device above. If those are present, yellow means the driver isn't correctly installed and red means hardware failure of the Prop Plug.

If other USB devices operate correctly, assume the trouble is your Prop Plug driver and reload it from here: <http://www.parallax.com/usbdriers>

When you can connect the Prop Plug and just see a COM port assignment without any status icons, note the COM port as that will be needed to tell the terminal software which device to communicate through. In this guide, COM 9 will be used.

You can optionally change the COM port assignment in the Device Manager screen. Select the device port you want to change, right click on it to access its properties and navigate to the Port Settings tab. Select the advanced button to see the port assignment screen shown below and input your desired COM port.



If you see a “port in use message”, you may have assigned multiple devices to the same port, and or may require a restart of your computer to fully assign the port. If you have trouble after a port assignment, try another port or use the default port assignment.

You are now ready to connect to the Propeller 2 chip.

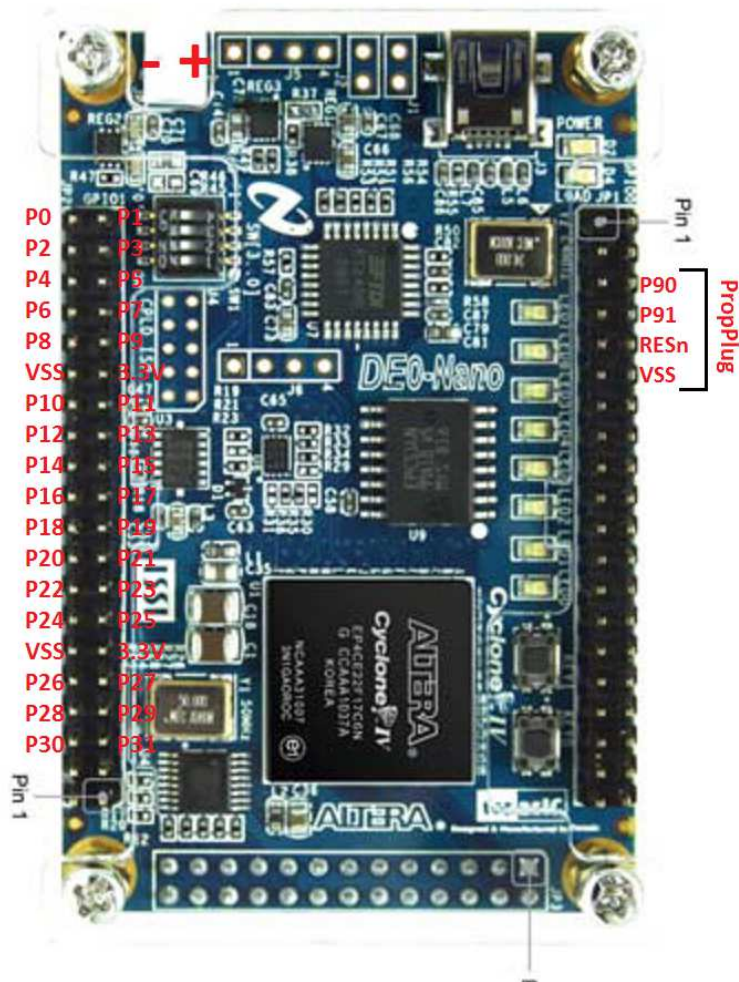
Connect Prop Plug to Emulated Propeller 2

Connect your Prop Plug to the Propeller 2 board. At the time of this writing, only ALTERA FPGA Propeller 2 emulation boards are available. This section will be revised and some content added to the document overall when that changes.

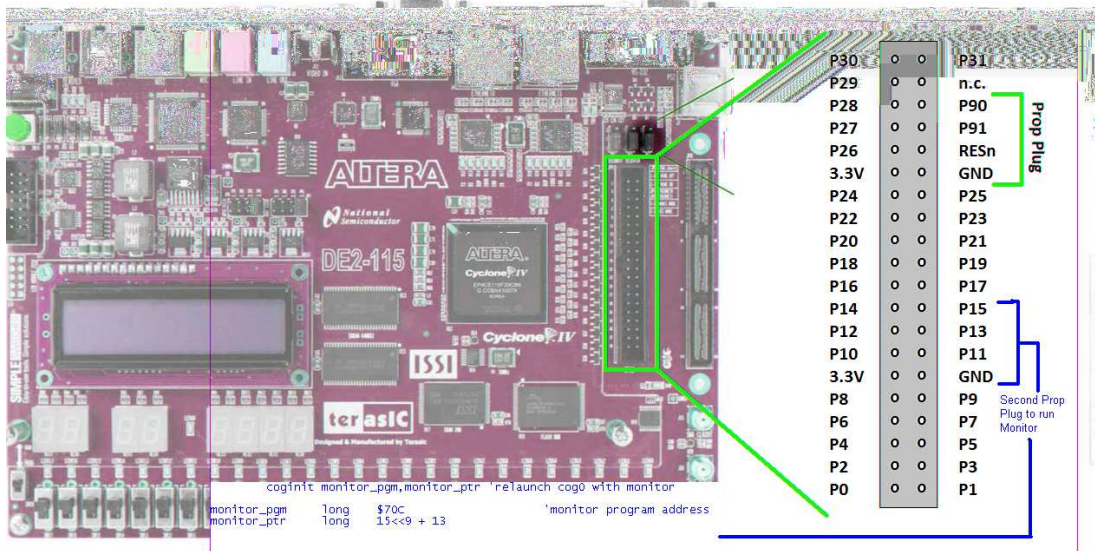
Two FPGA emulation kits are available: DE2-115 and DE0-NANO. The DE2 emulation supports 6 COGS and has LED connections and switches available for testing and the DE0 board supports 1 COG. This document was authored using the DE2 board. Not all examples shown are possible with the DE0 due to the lack of concurrent COGS in that emulation.

Additionally, this document does not cover the specifics of the FPGA emulation setup as that is temporary and subject to change when real hardware becomes available. Again, this document will be revised when that occurs. Until then, it's useful to understand how the monitor works and a lot of things you can do with it now on an emulated P2.

Here is a connection diagram for the DE0-NANO emulation:



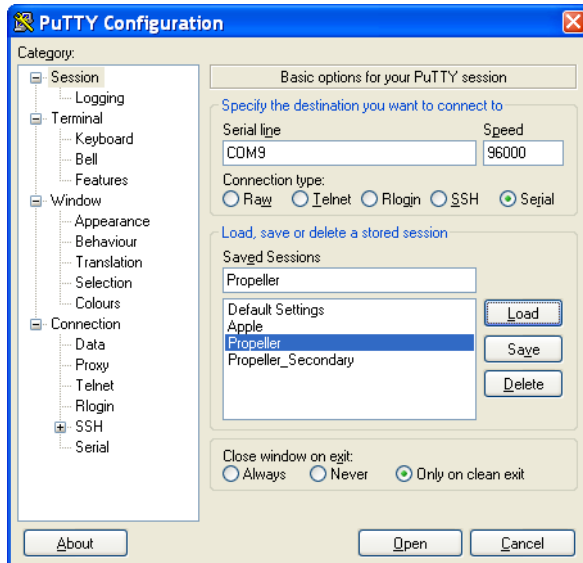
And this is the one or two Prop Plug connection diagram for the DE2-115 emulation:



Configure PUTTY

PUTTY requires very little configuration to operate well with the monitor. The key to using PUTTY quickly is to configure a setup and save it for future use. A two Prop Plug configuration will be shown, with COM 9 being the main Prop Plug, used for both programming the Propeller 2 and monitor communications, and COM 8 being the secondary Prop Plug used to access the monitor after a user loaded program calls the monitor on the secondary communication pins designated.

A minimum PUTTY setup looks like this:



Specify the COM port, a baud rate and select the “serial” button.

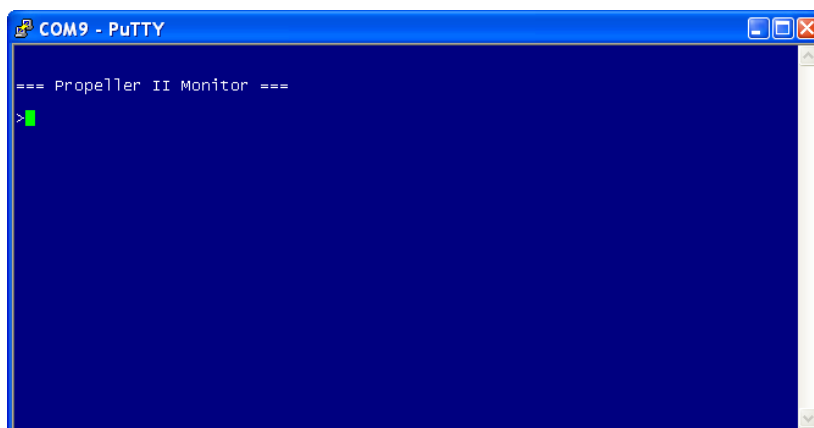
PUTTY requires that you populate the setup values, assign a name to them then select the “save” option to preserve them for future use.

If you don't do the save step, PUTTY will launch with those values and forget them when your session is done. This is good for testing.

Many options are available, such as colors, system bell, etc... The minimum ones needed to interact with the monitor are shown on this setup screen.

Power Up Quick Test

It's time to power up your Propeller 2 board and access the monitor. Make sure there are no bootable devices attached to your Propeller 2, turn on the power, launch PUTTY or your preferred terminal emulation software and press the space bar once to see the monitor reply:



Success! Now it's time to start using the monitor.

Using the Monitor Basics

Now let's take a tour of the P2 chip using the monitor. Despite the fact that no program has been loaded, there are a number of things to do at this early stage. From this stage onward, monitor screen shots will only be used when there is some value in doing so. For most use case explanation, plain text works. A different font will be used to differentiate monitor interaction text from the explanatory text, along with color used to highlight specific things..

The monitor prompt is a single ">" sign. When that is present, you can input commands. When it's not, the monitor is either not running or busy processing some command already given.

Built in help screen

Input a question mark and the CR / Enter key to see the built in command help text displayed on your terminal:

```
=== Propeller II Monitor ===
>?
                                     - HUB -
{adr{.adr}}                          - View
{adr{.adr}}/{dat{ dat}}              - Search
{adr{.adr}}:{dat{ dat}}              - Enter
adr.adr[</>]adr                       - Move
adr.adr^                              - Checksum
adr@                                  - Watch
[Y/W/N]                               - Byte/word/long
                                     - COGS -
cog+adr{+adr}                        - Start
cog-                                  - Stop
M                                    - Map
                                     - PINS -
{pin}[H/L/T/Z/R]                    - High/low/toggle/off/read
pin#                                  - Watch
pin|cfg                               - Configure
dat\                                  - Set DACs
                                     - MISC -
dat*                                  - Set clock
v                                    - Repeat
Q                                    - Quit
>
```

For clarity, the monitor prompt is colored blue, your input green and the output text red. Despite the small amount of text output by necessity so as to keep the Propeller 2 ROM size at a minimum, there is a lot of information on this simple reminder screen. In fact, it's the entire command reference for the monitor!

Command Groups

Commands are broken down into four groups based on what areas of the Propeller 2 they impact the most.

HUB

Hub commands operate on the shared HUB memory space, both RAM and ROM. To the monitor, there is no difference between the two. It's all HUB memory. To us, there is a difference in that the ROM is read only, and the RAM is writable. HUB commands can display memory contents for us, write values, search for occurrences of unique values, watch memory addresses and declare the unit display, such as bytes, words and longs.

COGS

Cog commands operate on COGS. It's possible to stop, start and map the state of the Propeller 2 COGS, even while a program is running!

PINS

These commands affect the state of the I/O pins. It is possible to set pin states, such as high, low, toggle, off, read; watch pins, configure them and set the DAC values associated with pins.

MISC

These commands are grouped because they don't fit well into the other categories. Miscellaneous commands include being able to set the clock, repeat input data and exit the monitor.

Basic Task Examples

Of these groups, the HUB commands are probably the easiest, so I'll start with those first. Feel free to follow along on your P2. Rather than write up an exhaustive detail on syntax, an interactive task based approach will be used. The syntax isn't difficult and should become obvious once you have worked through a few of these tasks.

Display Contents of HUB Memory

You can start out by just typing in an address. The monitor expects hexadecimal addresses, which consist of the numbers 0-9, and the letters A-F and addresses are not case sensitive, nor are they required to contain leading characters. If you are not familiar with hexadecimal addressing, Wikipedia, <http://en.wikipedia.org/wiki/Hexadecimal> along with many other sources have tutorials you can use.

For windows users, the system calculator has a programmer mode that includes quick and easy hex to decimal conversion. Select "view" from the calculator menu to access the scientific or programmer mode, depending on what version of Windows you are using. A tutorial on that can be found here: <http://grok.lsu.edu/article.aspx?articleid=8220>

Let's start with very low memory:

```
=== Propeller II Monitor ===
```

```
>4
00004- 32  '2'
>a
0000A- 7C  '|'
>A
0000A- 7C  '|'
>0.f
00000- 50 72 6F 70 32 2E 30 20 00 20 7C 0C 03 CA 7C 0C  'Prop2.0 . |...|.'
>
```

Again, the same conventions are used: Blue for the prompt, your input in green and the monitor output in red.

Going from top to bottom, the first address input was "4". The monitor output confirms the address in 5 digit form, followed by its contents as both a hex value 32 and an ASCII character, "2".

```
00004- 32  '2'
```

The next address input was "a", which is 10 decimal, and the value returned was 7C, character "|". Notice the third input "A" and that there is no difference in the output.

```
>a
0000A- 7C  '|'
>A
```

```
0000A- 7C  '|'
```

Remember, addresses are not case sensitive.

The final input is an address range! Two addresses separated by a period “.” will cause the monitor to display the full range of addresses. In this case, address 0 through F is 0 through 15 decimal, the first 16 addresses in the Propeller 2.

```
>0.f
00000- 50 72 6F 70 32 2E 30 20 00 20 7C 0C 03 CA 7C 0C  'Prop2.0 . |...|.'
```

These outputs feature some delimiters, highlighted in black above. Whenever the monitor outputs memory contents, it states the start address for the line, then one or more values, up to 16 per line separated by spaces. An ASCII text representation inside of single quotes follows to complete the line.

You read this as: Address 00000 = 50, address 00001 = 72, and so on until the end of the line, address 0000F = 0C.

Now let’s look at the end of ROM.

```
>E70.E8F
00E70- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D  '== End of ROM =='
00E80- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  '.....'
```

Note the series of text characters, “== End of ROM ==” form a useful marker to help you see where ROM ends and RAM begins.

When working with the monitor, it’s just simple letters, numbers and characters. Really, the bare minimum needed to get stuff done. When we write addresses formally, a very common convention for hex addresses is to precede them with a dollar sign to differentiate them from decimal numbers. Until this point in the guide, that has been ignored to keep things simple. Dollar sign hex addresses will be used in the explanatory text from now on where it makes sense to do so.

The end of ROM is \$0E7F, and the beginning of RAM is \$0E80. If we want to store values in the HUB, we must do that with addresses that are in the RAM region, which runs from \$0E80 to \$1FFFF. Notice the highest HUB memory address is 5 digits? That’s why the monitor formats all address output to 5 digits. You don’t have to worry about that on input however. When the monitor does it, really this just makes everything look nice and easy to read on the terminal.

One thing we can see from this simple HUB memory display is the RAM memory contents are all zeros! HUB memory is cleared during the Propeller boot process and we can see the results of that here.

Modify Contents of HUB Memory

Modifying HUB memory values works a lot like displaying HUB memory does.

Let’s say you want to put the number \$FF into address \$E80, the first available one. All you need to do is input the address, a colon and the value desired.

```
>e80:ff
```

Notice there is no output on this one. Technically, a write operation doesn’t yield output. The write just happens, if possible.

Here are two more:

```
>e83.e8f:ff
>e83.e8f:AA
```

These are simple range fills. First, the address range from \$E83 to \$E8F is filled with the value \$FF, then filled again with \$AA.

Now display to verify the results are as expected:

```
>e80.e8f
00E80- FF 00 00 AA AA AA AA AA AA AA AA AA AA AA AA '.....'
```

It's also possible to populate address sequentially:

```
>E90: 11 22 33 44 55 66 77 88 99 00
>E90.E9f
00E90- 11 22 33 44 55 66 77 88 99 00 00 00 00 00 00 '."3DUfw.....'
```

Patterns, Block Moves and Search

The monitor can either look through an address range for a specific pattern of values, or fill a range with a specific pattern of values, or block move the contents of HUB memory.

Write Multiple Values to an Address Range

Let's do reverse order this time. First, a fill address range with pattern. That end of ROM message is nice.

```
>e70.e7f
00E70- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D '== End of ROM =='
>1000.103f: 3d 3d 20 45 6e 64 20 6f 66 20 52 4f 4d 20 3d 3d
>1000.103f
01000- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D '== End of ROM =='
01010- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D '== End of ROM =='
01020- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D '== End of ROM =='
01030- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D '== End of ROM =='
>
```

The only difference here is simply supplying more values to the range fill operation. That is the pattern. Here is another example:

```
>2000.20ff: 00
>2000.2024: 11 22 33 44 55
>2000.2024
02000- 11 22 33 44 55 11 22 33 44 55 11 22 33 44 55 11 '."3DU."3DU."3DU.'
02010- 22 33 44 55 11 22 33 44 55 11 22 33 44 55 11 22 '"3DU."3DU."3DU."'
02020- 33 44 55 11 22 '3DU.'"
>
```

First zero the address range \$2000 - \$20FF. Then pattern fill \$2000 - \$2024 with the byte values, \$11, \$22, \$33, \$44, \$55, and finally display them to verify what happened.

Block Move Linear Chunks of RAM

Instead of typing all those in, just ask the monitor to move them instead! You use the greater than and less than characters to indicate which direction the move is to happen. In the example below, the address range containing that End of ROM string is copied to the destination address \$2000.

```
>e70.e7f
00E70- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D '== End of ROM =='
>e70.e7f>2000
>2000.201f
```

```

02000- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D      '== End of ROM =='
02010- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      '.....'

```

Much shorter and faster! Moves also come with no worries about fat fingering one of the values at the keyboard either. In this context, just like in Propeller Assembly Language, move really means copy.

The greater than character ">" means, block copy the address range (\$E70 - \$E7f) starting at the destination address. (\$2000)

Here is another move example:

```

>1000.2000:0
>1000.101f<0
>1000.102f
01000- 50 72 6F 70 32 2E 30 20 00 20 7C 0C 03 CA 7C 0C      'Prop2.0 . |...|.'
01010- 45 FE C1 0D E3 B6 FC 0C 01 C4 7C 0C 01 C4 7C 0D      'E.....|...|.'
01020- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      '.....'
>

```

In this example, the range \$1000 - \$2000 is filled with zeroes. The move is specified reversed from the example above.

```
>1000.101f<0
```

This means "fill the range \$1000 to \$101f with sequential and incrementing addresses beginning at \$0." Unlike the ">" example above, the range is now a target instead of a source, which is specified by the "<" character shown.

Finally, that range is displayed to verify the move was the one intended. "Prop2.0" is found at the very beginning of ROM, indicating the right move did happen.

Perform a Search within Data Range

You can quickly find where a specific set of data values appears in the HUB memory space. Here is a quick search for the word "ROM":

```

>0.e80/52 4f 4d
00E7A- 52 4F 4D      'ROM'
>

```

Sometimes there may be multiple occurrences of the target data string. The monitor includes a repeat command, a single quote "" and the colon ":" for this purpose. You start a search by the usual address range input shown above, then use the colon to continue with next address and the desired data followed by repeatedly using the single quote key to display multiple occurrences one at a time:

```

>100.e80/3d 3d
005AB- 3D 3D      '=='
>:/ 3d 3d
005C4- 3D 3D      '=='
>:/ 3d 3d
00E70- 3D 3D      '=='
>:/ 3d 3d
00E7E- 3D 3D      '=='
>:/ 3d 3d
>

```

The repeat character does not display. Instead, it just puts your input back on the line to process again quickly. For the search above, there were four occurrences. If you follow that example and continue to hit repeat, the search will begin again and display the four occurrences' repeatedly. This is useful if you are expecting some data to change, or you want to glance through the various occurrences again.

Plain Text Input

Use the single quote character to denote text input as shown. First, “now is the time” is written as ASCII text starting at \$2000, and then addresses \$2000 - \$200F are displayed to show the text in the HUB. Finally, a search is run from \$1500 - \$2200 for the string “is the”, which is found at \$2004.

```
>2000:'now is the time'  
>2000.200f  
02000- 6E 6F 77 20 69 73 20 74 68 65 20 74 69 6D 65 00   'now is the time.'  
>1500.2200/'is the'  
02004- 69 73 20 74 68 65   'is the'  
>
```

You only need the leading “ ‘ ” single quote character when the text is the only input; otherwise use two of them if the input is mixed hex and text:

```
>2000:ff 'hello' aa bb  
>2000.200f  
02000- FF 68 65 6C 6C 6F AA BB 00 00 00 00 00 00 00   '.hello.....'  
>
```

Watch Pins, Addresses and Set Various States

The monitor can watch a pin state change, or watch address values change. You can also set pin states and DAC Values.

Watch an Address

When running the monitor in tandem with other processes, it’s often useful to observe HUB memory values change. The watch command “@” does this.

Here is a short sample watch session:

```
>1000@  
50 00 02 04 FF  
>
```

In another instance of the monitor running on a different COG and pins, I changed the contents of \$1000 a few times so that there was something to watch:

```
>1000:00  
>1000:02  
>1000:04  
>1000:ff  
>
```

Once a watch is initiated, the monitor returns the current value of the address right away. When the monitor sees the value change, it outputs it. Press any key to end the watch and return the monitor prompt.

It’s possible that the values change more quickly than both the monitor can watch, and your serial device can display. Watch will simply return what it sees.

See Appendix B for another address watch example.

Configure I/O Pin and Set Pin State

The full details of I/O pin configuration are beyond the scope of this monitor reference guide; however, the simple example of setting a pin high or low will demonstrate how to use the command. I/O pins have many configuration options documented elsewhere. At this time, there is community reference documentation available here:

[The unofficial P2 documentation project - Google Drive](#)

To set a given pin, you specify the pin number in hex, followed by its configuration value. A configuration value of 0 sets the pin to be a simple logic output.

Pin numbers and configuration values are HEX values.

On the DE2, pins **\$20 (32)** to **\$31 (49)** are connected to the onboard LED's. For the NANO, you need to connect an LED through a suitable current limiting resistor or view the pin state with your meter or scope.

Let's set pin **\$21 (33)** to output and set it high, then low, then high:

```
>21|0
>21H
>21L
>21h
```

There is no monitor output on this command. Verify the pin status visually with an LED, meter or scope.

The first line configures the pin with two arguments separated by the pipe "|" character. "**21|0**" reads as "**configure pin \$21 to I/O configuration state 0**"

Once a pin has been configured, you can then set its state as shown, "**H**" for high, "**L**" for low. See the documentation for more information on other pin states possible.

Watch an I/O pin

In addition to watching addresses, the monitor can watch an I/O pin for you. This is useful for quick notification that the pin state is changing when there isn't an external indicator to use. As with addresses, the monitor will only report what it sees and it may not see every pin state change if the state changes are faster than the monitor can scan the pin.

In this example, any pin state change can be used. On the DE2, there are pushbuttons mapped to I/O pins \$32 (50) - \$34 (52). For the NANO, any of the I/O pins may be used with a suitable current limiting resistor.

Here is a sample watch session. It's very simple, input pin number and "#" and when you hit enter watching begins! Any key ends the watch.

```
>32#
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
>33#
1 0 1 0 1 0 1 0 1
>
```

First watch pin \$32, press the "Key 0" button a few times, end the watch. Watch pin \$33, press button "Key 1" a few times, end the watch.

SET COG DAC Values

Each COG has 4 hardware DACS. Emulation hardware is not yet available for testing. This command sets all four DACS to 8 bit values:

```
>115CCFF\
>
```

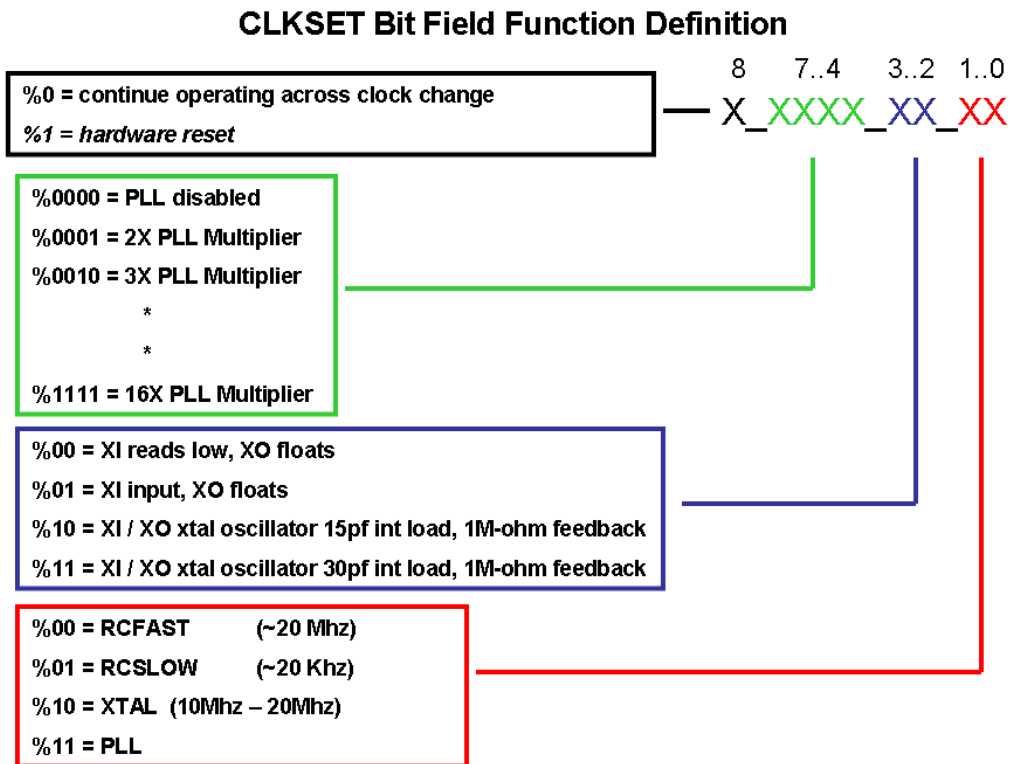
DAC 0 = FF, DAC 1 = CC, DAC 2 = 55, DAC 3 = 11

More information on the DAC values will be provided in the future.

Set the Propeller Clock (CLKSET)

This command takes a single data value and passes it directly to the CLKSET assembly language instruction, which then changes the Propeller 2 clock mode. Because this also involves establishing new baud rate metrics, you are prompted to hit the space bar so that communication with the monitor can continue with the new clock settings.

For reference, the CLKSET command takes 9 bits of data. (\$0 - \$1FF) The bit fields are:



Here is an example where the clock is first set to RCFAST, then RCSLOW:

```
>0*
Hit SPACE
>0.f
00000- 50 72 6F 70 32 2E 30 20 00 20 7C 0C 03 CA 7C 0C 'Prop2.0 . [...].'
>1*
Hit SPACE
>0.f
00000- 50 72 6F 70 32 2E 30 20 00 20 7C 0C 03 CA 7C 0C 'Prop2.0 . [...].'
>
```

More Advanced Examples

These examples all go beyond simple memory operations and into tasks that you may find useful when your programs are running or you are developing new tools.

Transfer small PASM program to HUB and run it on a COG

In this example, the Propeller 2 chip is running and the monitor is also running on one of the COGS. It is possible to directly transfer a PASM program to the Propeller 2 chip through the monitor and run it on one of the COGS without having to reset the chip, or utilize one of the Propeller loader utilities.

Here is a short PASM program that writes the counter to the output pins, which blink the LEDs in a binary pattern. On the DE2-115 board, those LEDs are connected to the Port B output pins. (P49..P32). The higher 8 bits are written so they can be seen blinking at a human scale rate.

```

' Terasic DE2-115 Prop2 Multi LED Blinker
' -----
DAT
                                org
                                mov    dirb, outmask    'make the port B I/Os outputs
:loop
                                getcnt A              'fetch lower 32 bits of global counter
                                shr    A, #16          'shift away the fast incrementing digits
                                and    A, outmask      'zero lower bits
                                mov    pinb, A         'write slow digits to LED's on DE2 board
                                jmp    #:loop          'keep doing it!
outmask
A                                long    $00_00_FF_00    'write higher bits only
                                res    1              'storage

```

Here is the object code listing from the Pnut.exe P2 compiler, with the program bytes of interest in red:

```

Object Code:
TYPE: 4B  VALUE: 00040004  NAME: LOOP[]
TYPE: 4B  VALUE: 00180018  NAME: OUTMASK
TYPE: 4C  VALUE: 001C001C  NAME: A

OBJ bytes:      28

_CLKMODE: 00
_CLKFREQ: 00B71B00

0000- 06 FA BF A0 0D 0E FC 0C 10 0E FC 28 06 0E BC 60  .....(....`
0010- 07 F2 BF A0 01 00 7C 1C 00 FF 00 00  .....|.....

```

Now it's time to enter the monitor, directly input this program and run it on one of the COGS.

```

=== Propeller II Monitor ===
>1000: 06 fa bf a0 0d 0e fc 0c 10 0e fc 28 06 0e bc 60
>1010: 07 f2 bf a0 01 00 7c 1c 00 ff 00 00

```

Display them to verify the correct bytes were entered.

```

>1000.101f
01000- 06 FA BF A0 0D 0E FC 0C 10 0E FC 28 06 0E BC 60  '.....(....`'
01010- 07 F2 BF A0 01 00 7C 1C 00 FF 00 00 00 00 00 00  '.....|.....'

```

Display COG State Map, Run the Program On Various COGS

The monitor can display the state of the COGS. On the DE2 board, 6 cogs are available. On the NANO, only a single COG is available.

Display COG Map to see monitor running on COG 0, other COGS free.

```
>m
0 0 0 0 0 0 M
```

Start up COG 2 with the Program found at \$1000

```
>2+1000
```

Map that to see COG 2 busy, and LED's blinking on the DE2.

```
>m
0 0 0 0 0 1 0 M
```

Start up COG 4 with the same program

```
>4+1000
```

Map now shows COG 2 and COG 4 active, monitor on COG 0

```
>m
0 0 0 1 0 1 0 M
```

Kill off both COGS. **Many commands can be stacked.**

```
>2-4-
>m
0 0 0 0 0 0 0 M
```

A COG number "+" program address starts a COG. A COG number "-" ends that COG. "M" displays the map.

Set BYTE, WORD, LONG modes

The Propeller 2 is a little endian CPU. *This means data is stored least significant bytes first, with more significant bytes stored sequentially toward higher RAM.* This can be very confusing and error prone on both input and display of data.

The monitor can operate in byte (Y), word(W) and long(N) modes to assist you.

Here is that same program listing at \$1000 shown in all three modes. *Notice you can stack commands here, declaring the mode right on the same line as an operation.*

```
>1000.101f
01000- 06 FA BF A0 0D 0E FC 0C 10 0E FC 28 06 0E BC 60   '.....(....`'
01010- 07 F2 BF A0 01 00 7C 1C 00 FF 00 00 00 00 00   '.....|.....'

>w
>1000.101f
01000- FA06 A0BF 0E0D 0CFC 0E10 28FC 0E06 60BC   '.....(....`'
01010- F207 A0BF 0001 1C7C FF00 0000 0000 0000   '.....|.....'

>N1000.101f
01000- A0BFFA06 0CFC0E0D 28FC0E10 60BC0E06   '.....(....`'
01010- A0BFF207 1C7C0001 0000FF00 00000000   '.....|.....'
```

Use Paste to Upload Programs and Data through Monitor

You may find it useful to upload data and programs into a running Propeller 2. The monitor can do this quickly and easily. All you need to do is send the data through the terminal emulation, or via the operating system into the monitor which will write it directly to the HUB for you.

You format the data the same way you would enter the data into a live monitor window.

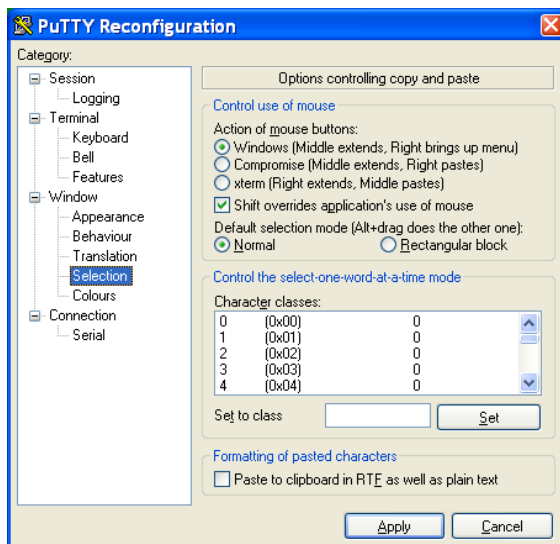
If you want to input words or longs, simply include the appropriate command in the data text. *Eg:* `n1000`: would specify long input starting at address \$1000.

Here is an example using the default byte input mode:

Copy this text into your clipboard, and optionally put it into a file named data.txt

```
1000: E0 06 FC 0C 04 06 3C 0C 03 00 7C 0C 0C 07 00 00
1010: FF FF FF FF FF
: 0D 1E 00 00 5B B4 00 00
103A: 11 22 33 44
: 55
: 66 77 88 99
1000: F0
```

Then paste into the terminal emulation:



Doing this varies according to your terminal software documentation. Some emulations allow a direct paste into the text window, others require configuration first. In PUTTY, your input device settings might require you declare what the paste action is. In the settings screen, the option "windows" is set so that a right click in the terminal window brings up "paste" as a menu option.

Here is the result of doing that paste: (paste text shown black)

```
=== Propeller II Monitor ===
>1000: E0 06 FC 0C 04 06 3C 0C 03 00 7C 0C 0C 07 00 00
>1010: FF FF FF FF FF
>: 0D 1E 00 00 5B B4 00 00
>103A: 11 22 33 44
>: 55
>: 66 77 88 99
>1000: F0
>1000.104f
01000- F0 06 FC 0C 04 06 3C 0C 03 00 7C 0C 0C 07 00 00
01010- FF FF FF FF FF 0D 1E 00 00 5B B4 00 00 00 00
01020- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01030- 00 00 00 00 00 00 00 00 00 00 11 22 33 44 55 66
01040- 77 88 99 00 00 00 00 00 00 00 00 00 00 00 00
>
```

Notice the monitor knows what the last address it operated on is. **Any line beginning with just a colon means “continue on using current address + 1”** That allows for irregular or large amounts of data to be uploaded without having to format each line individually with an address. Simply start with one address, then use the colon character “:” to continue data input from where ever the last line left off.

Input is up to 16 bytes at a time, due to the line input limit built into the monitor itself.

Data upload starts with some byte data input at address \$1000. The next line explicitly specifies the address \$1010, which is also the next address to accept data. This alignment is nice, but not necessary at all. Notice the address \$1000 is specified again in the last line of the pasted text! That value, \$F0 overwrites the original value of \$E0 placed at the start of the operation.

Use explicit addresses when you want to initiate a block of data, then use the colon “:” to continue to supply data allowing the monitor to automatically increment the destination address for you. This way, it’s possible to lightly edit an input file to place data somewhere else in memory quickly and easily.

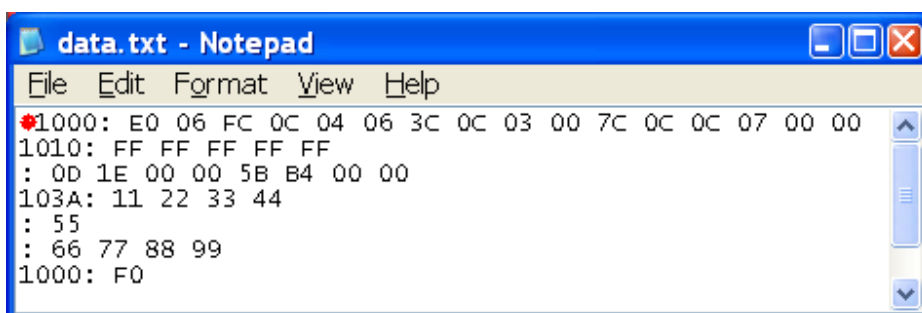
Note the colon following address \$1010 and compare the bytes to the memory dump to see that the line with the colon simply means, “Next address” If the address \$103A were to be changed to \$2000, those 4 bytes on that input line along with the five bytes on the next two lines would be located starting at \$2000 with only that one change required, because the colon works relative to the last explicit address given to the monitor.

You can perform a memory display or have the monitor compute a checksum to verify all lines input correctly.

Copy Data File into Propeller 2 through Monitor

If it works with a paste operation, it will also work with a simple file copy to the serial device. A Windows example will be shown. UNIX works in a similar way, but you will have to identify your serial device table entry for use with the “cat” command. (`cat data.txt > /dev/ttyd5` for example)

Here is the same example text placed into a file:



The red dot indicates a single space is in the file, because that is the character the monitor uses for its auto baud rate function. There is a single new line at the bottom to complete the input of the last line as well.

Copy this file to the COM port you know is connected to the monitor using the copy command in a standard Windows command prompt:

```
Microsoft windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

```
C:\users\parallax>copy data.txt com8
1 file(s) copied.
```

```
C:\users\parallax>
```

You may find it necessary to change directory or drive or type a longer path to properly identify your data file. If there are two monitors running, simply use one to verify the data input correctly, or when the copy is complete, connect your terminal to the COM port to interact with the monitor directly. It is still there waiting for more input.

```
@
? - Help
>1000.104f
01000- F0 06 FC 0C 04 06 3C 0C 03 00 7C 0C 0C 07 00 00   '.....<...|.....'
01010- FF FF FF FF FF 0D 1E 00 00 5B B4 00 00 00 00 00   '.....[.....'
01020- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   '.....'
01030- 00 00 00 00 00 00 00 00 00 00 00 11 22 33 44 55 66   '....."3DUf'
01040- 77 88 99 00 00 00 00 00 00 00 00 00 00 00 00 00   'w.....'
>
```

Switching like this live sometimes takes a bit of practice. I find inputting a space character or two, followed by a carriage return generally gets me the system bell, followed by a prompt. Once I see the prompt, the monitor has adapted to the new connection and can then display results. Be careful not to input addresses, or if you do, make sure they are low read only addresses so that data isn't overwritten by mistake.

You may also find using the windows "type" command more effective depending on whether or not you want to send multiple files or make use of the operating system redirection and or pipe capability: "type data.txt > com8"

Compute a Checksum

To do this with the monitor, specify an address range followed by the caret "^" character:

```
>1000.104f^
00000c20
```

The monitor computes a standard 32 bit sum of the values contained in the address range specified. Here is the PASM code listing for the checksum command routine, with some added comments:

```
cmd_checksum      call    #check_range          'check range is valid

:loop             setptr  v1                    'prepare to sum all the bytes
                  call    #rdxxxx              'get a byte, returned in value
                  add     y,value              'accumulate sum in y
                  djnz    z,#:loop            'Do all the bytes

                  mov     value,y              'prepare to print sum (value)
                  mov     hsize,#8             'number of digits to return to user
                  call    #tx_hex              'send digits over serial

                  jmp     #cmd_next_cr1f      'next command
```

Launch Monitor From Within Your Program

This is useful on both the single COG NANO as well as the DE2 multi-cog FPGA emulations. Since the NANO only offers a single COG, it's very useful to end a program with a call to the monitor for debugging and examining results! On the DE2, the monitor can run alongside your program allowing for inspection of data, starting and stopping COGS and or modifying values interactively.

Starting the monitor from within your own program is simple. All you need to do is specify the monitor start address in the HUB, which is \$70C by default when you start it from the ROM, your serial I/O pins for TX / RX, and optionally a COG for it to run on. You can include the PASM instructions and data below in the template listing into your own program. Simply modify the starting COG value and tx/rx pin definitions.

Here is a commented program template listing for reference:

```

DAT
start_mon      org
                setcog #n '+ %1000          'uncomment '+ %1000' for next available cog
                coginit monitor_pgm, monitor_ptr 'function; otherwise,
                'start monitor on cog n

monitor_pgm    long   $70C                'ROM entry point for monitor,
monitor_ptr    long   90<<9 + 91        'this could be RAM for a modified monitor.
                'serial pins = RX<<9 + TX

```

The SETCOG instruction controls how the COGINIT instruction does things. If you specify a valid COG number, known as a COG ID between 0 and 7 (%000 to %111 binary), COGINIT will start that COG with the monitor. COGINIT can even restart the COG it is running in with a new program!

When SETCOG is given a value that results in bit three being set (%1xxx), the behavior of COGINIT changes to start the monitor on the next available COG, not a specific COG.

This functionality is important on the NANO FPGA emulation because it only has one COG to operate with, which requires COG 0 be specified as the monitor COG. On the DE2, 6 COGS are available, meaning you can choose to let the Propeller 2 assign the next free COG, or choose one from the 6 available.

The other two lines, labeled “monitor_pgm” and “monitor_ptr” hold the monitor HUB start address and pin definitions needed for the COGINIT command to do its job.

You are free to run multiple instances of the monitor as long as there are COGS free to run them and unique pins available for each monitor to communicate on.

Please see the program listing “Running Multiple Monitors” in Appendix B for more information.

Modify Running PASM Program

This example demonstrates the modification of HUB run-time values while a program is running, and the replacement of a COG program with a different one, also while a program is running.

This example is kept simple, limited to blinking LED's and one of the pushbuttons on the DE2 emulation board.

For now, until the state of tools has advanced, Pnut.exe assembles things at \$0, but the program load happens at the beginning of RAM. This means HUB address references need to have \$E80 added to them, or they will be referencing values in the ROM. This information is presented as a means to understand the very basic things that happen with a P2 is loaded with a program.

Please refer to the program listing in Appendix B, “Replace_Example.spin” throughout this example.

Here is the object code listing from Pnut.exe: (load .spin file, then ctrl-L)

```

TYPE: 4B VALUE: 00000000 NAME: START_MON
TYPE: 4B VALUE: 00140014 NAME: LOOP
TYPE: 4B VALUE: 002C002C NAME: MONITOR_PGM
TYPE: 4B VALUE: 00300030 NAME: MONITOR_PTR
TYPE: 4B VALUE: 00340034 NAME: WRITE_ADDRESS
TYPE: 4B VALUE: 00380038 NAME: ENTRY_ADDR
TYPE: 4C VALUE: 003C003C NAME: A
TYPE: 4B VALUE: 0000003C NAME: ENTRY
TYPE: 4B VALUE: 00080044 NAME: LOOP
TYPE: 4B VALUE: 00240060 NAME: PIN
TYPE: 4B VALUE: 00280064 NAME: DELAY
TYPE: 4B VALUE: 002C0068 NAME: DELAY_ADDRESS
TYPE: 4B VALUE: 0030006C NAME: TOGGLE_MASK
TYPE: 4C VALUE: 00340070 NAME: TIME

```

OBJ bytes: 112

```

_CLKMODE: 00
_CLKFREQ: 00B71B00

```

```

0000- E0 06 FC 0C 0C 16 3C 0C E0 10 FC 0C 0F 1C 3C 0C .....<.....<.
0010- 01 FA FF A4 0D 1E FC 0C 18 1E FC 28 08 1E FC 2C .....(.....)
0020- 0F F2 BF A0 0D 1E 3C 04 05 00 7C 1C 0C 07 00 00 .....<...|.....
0030- 5B B4 00 00 00 20 00 00 BC 0E 00 00 DB 12 7C 0C [...|.....]
0040- 09 18 BC 2C 0D 1A FC 0C 0A 1A BC 80 0F 1A FC 80 .....
0050- 0A 1A BC FC 0C F2 BF 6C 0B 14 BC 08 02 00 7C 1C .....|.....
0060- 21 00 00 00 80 C3 C9 01 E4 0E 00 00 01 00 00 00 !.....

```

And a monitor memory dump of the program after it has been loaded and executed:

```

00E80- E0 06 FC 0C 0C 16 3C 0C E0 10 FC 0C 0F 1C 3C 0C '.....<.....<.'
00E90- 01 FA FF A4 0D 1E FC 0C 18 1E FC 28 08 1E FC 2C '.....(.....)'
00EA0- 0F F2 BF A0 0D 1E 3C 04 05 00 7C 1C 0C 07 00 00 '.....<...|.....'
00EB0- 5B B4 00 00 00 20 00 00 BC 0E 00 00 DB 12 7C 0C '[.....|.....]'
00EC0- 09 18 BC 2C 0D 1A FC 0C 0A 1A BC 80 0F 1A FC 80 '.....'
00ED0- 0A 1A BC FC 0C F2 BF 6C 0B 14 BC 08 02 00 7C 1C '.....|.....'
00EE0- 21 00 00 00 80 C3 C9 01 E4 0E 00 00 01 00 00 00 '!.....'

00E80- 0CFC06E0 0C3C160C 0CFC10E0 0C3C1C0F '.....<.....<.'
00E90- A4FFFA01 0CFC1E0D 28FC1E18 2CFC1E08 '.....(.....)'
00EA0- A0BFF20F 043C1E0D 1C7C0005 0000070C '.....<...|.....'
00EB0- 0000B45B 00002000 00000EBC 0C7C12DB '[.....|.....]'
00EC0- 2CBC1809 0CFC1A0D 80BC1A0A 80FC1A0F '.....'
00ED0- FCBC1A0A 6CBFF20C 08BC140B 1C7C0002 '.....|.....'
00EE0- 00000021 01C9C380 00000EE4 00000001 '!.....'

```

This program starts up three COGS. Here is the COG map:

```

>m
0 0 0 0 M 0 1 1

```

An instance of the monitor is running on COG 3, and the two PASM programs running on COGS 0 and 1. COG 0 contains the counter to led blinker, and COG 1 contains the single pin blinker.

A counter value is being written to HUB address \$2000, and a blink delay value is being fetched from HUB address \$EE4, highlighted in blue above. Other addresses contain constants that get copied into the COG, and are not directly modifiable without restarting the COG with a new value.

Some of the byte address offsets in hex are highlighted above. Notice at the start of the program, the number of bytes from \$0 and the COG ORG offset bytes are the same. At label “entry”, a new ORG directive takes effect, with the lower order word absolute and the higher order word “cog relative” due to the influence of the ORG directive.

You can use either the offset values or a simple numerical search and or instruction long counts to locate where data / addresses are in the object code, or simply label more of them as needed.

“write_address” = \$0 + \$34 + \$E80 = \$Eb4 contains address value \$2000

“pin” = \$0 + \$60 + \$E80 = \$EE0 = contains pin value \$21 (33)

“delay” = \$0 + \$64 + \$E80 = \$EE4 = contains delay value \$01C9C380 = 30_000_000

The higher word in the object code listing contains the offset from the last ORG directive.

“Entry” = \$0 + \$3C + \$E80 = \$EBC = Base address of second PASM program on COG 1.

“Pin” = \$EBC + \$24 = \$EE0 = delay value...

Now we know where all the addresses are and can now use the monitor to do things.

First, the easy one, let's watch the counter value written to the HUB:

```
>2000@
00002E00 00002F00 00003000 00003100 00003200 00003300 00003400 00003500 00003600 00003700
00003800 00003900 00003A00 00003B00 00003C00
```

Next, modify the blinker time delay value at address \$EE4, first a really slow blink followed by a faster one:

```
>ee4
00EE4- 01C9C380  '....'
>ee4: 0FFF0000
>ee4: 00ff0000
>
```

Let's say we want to change the blinking pin. That's a COG value; meaning COG 1 needs to be stopped and started again with the new value, pin \$20 (30) this time.

```
>ee0: 20
>ee0
00EE0- 00000020  '... '
>1-
>1+ebc
>
```

Finally, modify the COG 0 program to stop writing values to the HUB. One easy way to do that would be to simply insert a NOP instruction (\$0) where the WRWORD instruction currently is and restart the COG. A quick look at the program listing:

```
:loop
    getcmt A          'fetch lower 32 bits of global counter
    shr    A, #24      'shift away the fast incrementing digits
    shl    A, #8
    mov    pinb, A     'write upper counter digits to LED's
    wrword A, write_address 'Put a value in the hub to watch with monitor
    jmp    #:loop     'keep doing it!

monitor_pgm long $70C 'ROM entry point for monitor,
```

shows us the WRWORD instruction simply is two longs lower in HUB memory than the data stored at label “monitor_pgm”

The target address to zero out is: \$0 + \$2C – (two instructions = \$8) + \$E80 = \$EA4

Now we zero the instruction out, stop the COG and restart it and verify nothing is being written to hub location \$2000:

```
>ea4:0
```

```
>0-
>0+e80
ÿ
=== Propeller II Monitor ===
>m
 0 0 0 0 M 1 1 1
>N2000@
00003800
>
```

The monitor dies off as the COGINIT instructions are still at the beginning of the program, and get executed again when the COG is restarted. The solution is to either select another entry point to restart the COG, or replace those instructions with a NOP. Because the single pin blinker is initiated with “next available cog” (%1000) parameter value to COGINIT, there are now two of them running on COGS 1 and 2, as well, both attempting to blink the same pin.

Address \$2000 contains the last value written and isn't changing. Success!

Managing things from here is left as an exercise for the reader.

Closing

Ideally, you the reader now have a good understanding of how to use the monitor built into the Propeller 2 chip along with some basic, low level understanding of what happens when a program gets compiled and loaded into the Propeller 2.

Not all examples are intended as practical, every day production use cases. They do however present some options that may be handy in some scenarios. One such scenario would be building a program up in pieces, potentially loading data and other elements into the Propeller 2 memory map for testing and or capture as object code sans more advanced development tools.

It is also possible to do lots of rapid testing without power cycling the Propeller 2 as well. Programs as well as data can be dropped into place along side running ones using the HUB memory to communicate parameters and data between both the programs and the user via the monitor and terminal.

At the time of this writing, the Propeller 2 and development tools are both in early stages. When that changes, this document may well be expanded and modified to better serve you the reader.

From here it's on to bigger and better things to do with your Propeller 2!

Appendix A Propeller 2 ROM Program Listings

ROM_Booter

The booter is located first in HUB memory, starting at location 0. The first 8 bytes contain the Chip Version string information, "Prop2.0" followed by the shutdown routine used in case of unsuccessful boot. Location \$10 (16) is where the booter program starts. The booter runs on COG 0, launching SHA256 when needed on COG 1.

```
'*****
'*
'*          Propeller II ROM Booter          *
'*
'*          Version 0.1                      *
'*
'*          11/01/2012                      *
'*
'******
CON
    rx_pin = 91
    tx_pin = 90
    spi_cs = 89
    spi_ck = 88
    spi_di = 87
    spi_do = 86

    base = $E80

DAT
;
;
;  version (@$000)
;
;          byte    "Prop2.0 "
;
;
;  shutdown (@$008)
;
;          org
;
;          clkset offset+h001          'set clock to rc slow
;          cogstop offset+h200        'stop cog0
offset
;
;
;  Booter (@$010)
;
;          org
;
;          reps    #256,@:fuse        'ready to read 256 fuses
;          setport #rx_pin            'set rx_pin port for booting
;
;          cogid   fuse_read          nr    'read fuses (172 fuses + 84 zeros)
;          cogid   fuse_read          nr,wc '(last iteration initializes cnt to
;                                           '$00000000_00000001)
;
;          add     fuse_read,#1
;          test    fuse_read,#$1F wz
;:fusex          rcr     fuses,#1
;:fuse  if_z      add     :fusex,h200
;
;          cogid   spi_read          nr    'disable fuses and enable cnt
;                                           '(spi_read[10..0] = 0)
;
;
;
;  Attempt to boot from serial
;
;          jnp     monitor_ptr,#boot_flash  'if rx_pin is low, skip serial and
;                                           'boot from flash
```

```

        call    #rx_bit          'measure low rx calibration pulses
        mov     threshold,delta  '( host $F9 -> %1..010011111..)
        call   #rx_bit          'and calculate threshold
        add    threshold,delta  '(any timeout results in flash boot)
h001    shr     threshold,#1     '(9 lsb's are $001)

        mov     count,#250      'ready to receive/verify 250 lfsr bits
:lfsrin call   #rx_bit              'receive bit ($FE/$FF) into c
        test   lfsr,#$01        'get lfsr bit into nz
        jmp    #boot_flash      'if mismatch, boot from flash
        if_c_eq_z                'advance lfsr
        test   lfsr,#$B2        '
        rcl   lfsr,#1           '
        djnz  count,#:lfsrin    'loop for next bit in

        mov     count,#250+8    'ready to transmit 250 lfsr bits
:lfsrout cmp    count,#8        '+ 8 version bits
        if_z   mov     lfsr,#$52 'if last 8 bits, set lfsr so that version
        test   lfsr,#$01        'will be output
        call   #wait_rx         '$52 results in version $20 being sent
        clrp  #tx_pin          '(%00000100)
        call   #wait_rx         'get lfsr/version bit into nz, z=1 on
        setpnz #tx_pin         'last iteration
        call   #wait_rx         'wait for rx low (convey incoming $F9 on
        test   lfsr,#$B2        'rx_pin to $FE/$FF on tx_pin)
        rcl   lfsr,#1           'make tx low
        djnz  count,#:lfsrout   'wait for rx high
        jmp    #load            'make tx lfsr/version bit
        call   #wait_rx         'wait for rx low
        test   lfsr,#$B2        'make tx high
        rcl   lfsr,#1           'wait for rx high
        djnz  count,#:lfsrout   'advance lfsr
        jmp    #load            'loop for next bit out

        jmp    #load            'serial handshake done, attempt to load
        ,                      'from serial (z=1)
        ,
        ,
        ' wait for rx low/high - if timeout, attempt to boot from flash
wait_rx  getcnt  time           'ready timeout
        add    time,timeout

:waitpxx waitpne rx_mask,rx_mask wc    'wait for rx low/high with timeout

wait_rx_ret notb   :waitpxx,#23    'toggle waitpeq/waitpne
        if_nc  ret                'return if not timeout (boot_flash follows)
        ,
        ,
        ' Attempt to boot from flash
boot_flash mov    count,#4      'ready for 3 resets and 1 read command
:cmd     setp    #spi_cs         'spi_cs high
        clr    #spi_ck         'spi_ck low

        reps   #32,@:bit        'ready for 32 command bits
        clr    #spi_cs         'spi_cs low

        if_nc  cmpr   count,#1    wc    'first 3 commands = $FF_FF_FF_FF (reset)
        rol   spi_read,#1      wc,wz  'last command = $03_00_00_00 (read from 0),
        ,                      'z=0

        setpc  #spi_di          'cycle spi_ck
        setp  #spi_ck
        clr   #spi_ck

:bit     djnz   count,#:cmd      'loop for next spi command
        ,
        ,
        ' Load from serial (z=1) or flash (z=0)
load     setptr loader_pgm      'load loader into base+$000..$7DF, HMAC into
        ,                      'base+$7E0..$7FF

:long    mov     count,h200      'ready to input $200 longs
        mov    bits,#32        'ready to input 32 data bits

:bit     if_z   call   #rx_bit    'input serial bit (serial mode)
        if_nz  getp  #spi_do     wc    'input spi_do (flash mode)

```

```

if_nz setp #spi_ck 'high spi_ck (flash mode)
if_nz clr #spi_ck 'low spi_ck (flash_mode)
      rcl data,#1 'shift bit into long
      djnz bits,#:bit 'loop, adequate time for next flash bit

      wrlong data,ptr++ 'store long in hub ram
      ' (ptr=base+$800 after)
      djnz count,#:long 'loop for next long
,
,
Compute loader HMAC signature for loader authentication
,
base+$000..$7DF = loader ($1F8 longs)
base+$7E0..$7FF = loader HMAC signature (8 longs)
base+$800..$81F = fuses, 1st half are HMAC key (8 longs)
base+$820..$83F = proper HMAC signature (8 longs)
base+$840..$843 = sha256 command interface (1 long)
,
      reps #8,#1 'store 128-bit key + 44 extra
      'fuses + 84 zero bits
      setinda #fuses 'into base+$800..$81F
      wrlong inda++,ptr++ ' (ptr = base+$820, afterwards)

      setcog #1 'launch cog1 with sha256
      coginit sha256_pgm,sha256_ptr '(1st command will be
      'set before sha256 executes)

      setinda #begin_hmac 'do sha256 commands to compute
      'proper loader hmac
      mov count,#3 'ready for 3 commands: begin_hmac,
      'hash_bytes, read_hash
:cmd wrlong inda++,sha256_ptr 'set command
:wait rdlong data,sha256_ptr wz 'wait for command done
      tjnz data,#:wait
      djnz count,#:cmd 'loop for next command (z=1 after)

      cogstop h001 'done with sha256, stop cog1
,
,
If loader authenticates, run it
,
      reps #8,@:cmp 'verify loader hmac signature (z=1 on entry)
      setcog #0 'ready to relaunch cog0 with
      'loader/shutdown/monitor

      rdlong data,ptr[-$10] 'get loader hmac signature long
      rdlong bits,ptr++ 'get proper hmac signature long
:cmp if_z cmp data,#1 wz 'bits wz 'compare, z=1 if authenticated

      if_z coginit loader_pgm,loader_ptr 'if loader authenticated,
      'relaunch cog0 with loader
,
,
Authentication failed, hide fuses and clear memory
,
      reps #($20000/16),#1 'ready to clear all memory
      cogid monitor_pgm nr 'hide fuses (set bit 10)
      wrquad ptr++ 'clear 16 bytes at a time (quad=0)
,
,
If key < 0, shut down - else, start monitor
,
      if_z or fuses+0,fuses+1 wz 'check if 128-bit key = 0
      or fuses+2,fuses+3 wz
      if_nz mov monitor_pgm,#$008 'if key < 0, set shutdown,
      '(overwrites fuse data in cog regs)

      coginit monitor_pgm,monitor_ptr 'relaunch cog0 with shutdown
      'or monitor
,
,
Receive bit (c) - compare incoming pulse to threshold
rx_bit call #wait_rx 'wait for rx low
      getcnt delta 'get time

      call #wait_rx 'wait for rx high
      subcnt delta 'get time delta

      cmp delta,threshold wc 'compare time delta to threshold

```

```

rx_bit_ret    ret
,
' Constants
,
timeout      long    60_000_000/1000*150    '150ms @20MHz (rcfast)
spi_read     long    $03_000000
rx_mask      long    |< rx_pin
fuse_read    long    $200                    '(becomes $300)
h200         long    $200
lfsr         long    "P"

begin_hmac   long    0 '1<<30 + (($004<<2)-1)<<17 + base+$800
              'begin_hmac, loads key at base+$800 (4 longs)

hash_bytes   long    0 '2<<30 + (($1F8<<2)-1)<<17 + base+$000
              'hash_bytes, hashes message at base+$000 ($1F8 longs)

read_hash    long    0 '3<<30                    + base+$820
              'read_hash, writes hash at base+$820 (8 longs)

sha256_pgm   long    $1CC                    'sha256 program address
sha256_ptr   long    base+$840                'sha256 parameter (points to command)

loader_pgm   long    base+$000                'loader program address
loader_ptr   long    base+$800                'loader parameter (points to fuses)

monitor_pgm  long    $558+$1B4                'monitor program address
monitor_ptr  long    tx_pin<<9 + rx_pin        'monitor parameter (conveys pins)
,
' Variables
,
fuses        res     8
count        res     1
bits         res     1
data         res     1
time         res     1
delta        res     1
threshold    res     1

```

ROM_SHA256

This is the encryption routine used to validate incoming code against the programmable keys. It runs on COG 1 during the boot process.

```
*****
'*
'*          Propeller II ROM SHA-256/HMAC          *
'*          Version 0.1                            *
'*          11/01/2012                             *
'*
*****

;
; Usage:          commandlong := 0                  'pre-
; clear command long          cognew($1CC, @commandlong)      'start
; SHA-256/HMAC in new cog
; Start here for HMAC:      commandlong := 1<<30 + (keysize-1)<<17 + @key  'start
; HMAC with key of keysize bytes (1..64)          repeat while commandlong      '(wait
; for command done)
; Start here for SHA-256:   commandlong := 2<<30 + (msgsize-1)<<17 + @msg  'hash msg
; of msgsize bytes (1..8192)          repeat while commandlong      '(wait
; for command done)
;
; {issue more 2<<30 commands if msg > 8192 bytes}
;
; resulting hash into hashbuffer (32 bytes)      commandlong := 3<<30 + @hashbuffer      'read
; for command done)          repeat while commandlong      '(wait
;
; {hasbuffer now contains result, ready for new 1<<30 or
; 2<<30 command}

DAT

                org
                setf    #0_1111_0000 'configure movf for sbyte0 ->
{dbyte3,dbyte2,dbyte1,dbyte0,dbyte3,...}
                call    #init_hash   'init hash, clear hmac mode, reset byte count
;
; Command Loop
;
command         rdlong  x,ptr         'wait for command
(%cc_nnnnnnnnnnnn_pppppppppppppppp)
                tjz     x,#command
                setptrb x              'get pointer (%pppppppppppppppppp)
                mov     count,x        'get count (%nnnnnnnnnnnn)
                shl    count,#2
                shr    count,#2+17
                add    count,#1       '+1 for 1..8192 range
                shr    x,#32-2        'get command (%cc)
                cachex                'invalidate cache for fresh rdbytec's
1..64)         djz     x,#begin_hmac  '1 = begin hmac, pointer @key (count+1 bytes,
1..8192)       djz     x,#hash_bytes  '2 = hash bytes, pointer @message (count+1 bytes,
done           djz     x,#read_hash   '3 = read hash, pointer @hashbuffer (32 bytes)
                wrlong zero,ptr       'clear command to signal done
```



```

,
,
,
' Begin HMAC
begin_hmac    call    #end_hash        'end any hash in progress
:ipad        mov     x,#$00                'get and hash ipad key (full block)
              cmpr   count,bytes wc 'after key bytes, hash $00's to fill block
              if_c   rdbytec x,ptrb++
              xor    x,#$36                'xor bytes with ipad ($36)
              call   #hash_byte          '(last iteration triggers hash_block, z=1)
              if_nz  jmp     #:ipad

              reps   #16,#2            'save opad key
              setinds #opad_key,#w
              mov    indb,inda++
              xor    indb++,opad        'xor bytes with opad ($5C)

              mov    hmac,#1            'set hmac mode
              jmp    #done

,
,
' Hash Bytes
hash_bytes    rdbytec x,ptrb++        'hash bytes
              call   #hash_byte
              djnz   count,#hash_bytes

              jmp    #done

,
,
' Read Hash
read_hash     tjz    hmac,#:not        'if not hmac, output hash

              call   #end_hash        'hmac, end current hash

              reps   #16,#1            'get opad key into w[0..15] (full block)
              setinds #w,#opad_key
              mov    indb++,inda++

              call   #hash_block        'hash opad key

              reps   #8,#1             'get hashx[0..7] into w[0..7]
              setinds #w,#hashx
              mov    indb++,inda++

              movd   hash_byte,#w+8     'account for opad key and hashx bytes
              mov    bytes,#64+32      '(1-1/2 blocks, 1/2 block needs end_hash)

:not          call   #end_hash        'end current hash

              setinda #hashx
              mov    count,#8          'store hashx[0..7] at pointer, big-endian
:out          mov    #4,#2
              reps   #4,#2
              mov    x,inda++
              rol    x,#8
              wrbyte x,ptrb++
              djnz   count,#:out

              jmp    #done

,
,
' End Hash - hash $80, any $00's needed to get to offset $38, then 8-byte length
end_hash      mov    length,bytes      'get message length in bits
              shl    length,#3

:fill        mov    x,#$80                'hash end-of-message byte ($80)
              call   #hash_byte          '(may trigger hash_block)
              mov    x,bytes            'hash any $00's needed to get to offset $38
              and    x,#$3F
              cmp    x,#$38 wz
              mov    x,#$00
              if_nz  jmp    #:fill

```

```

:len      test    bytes,#$04 wc 'hash 8-byte length, big-endian
         if_c   rol     length,#8   '(hash four $00's, then four length bytes)
         if_c   mov     x,length
         if_nz  call    #hash_byte '(last iteration triggers hash_block)
         jmp    #:len

         reps   #8,#1      'save hash[0..7] into hashx[0..7]
         setinds #hashx,#hash
         mov    indb++,inda++

init_hash  reps   #8,#1      'copy hash_init[0..7] into hash[0..7]
         setinds #hash,#hash_init
         mov    indb++,inda++

         mov    hmac,#0     'clear hmac mode
         mov    bytes,#0    'reset byte count

init_hash_ret  ret
end_hash_ret   ret
;
; Hash Byte - add byte to w[0..15] and hash block if full (z=1)
hash_byte   movf    w,x      'store byte into w[0..15], big-endian
         add    bytes,#1    'increment byte count
         if_z   test    bytes,#$03 wz 'every 4th byte, increment w pointer
         add    hash_byte,d0
         if_z   test    bytes,#$3F wz 'every 64th byte, reset w pointer
         movd   hash_byte,#w
         if_z   call    #hash_block 'every 64th byte, hash block
hash_byte_ret  ret
;
; Hash Block - first extend w[0..15] into w[16..63] to generate schedule
hash_block  reps   #48,@:sch 'i = 16..63
         setinds #w+16,#w+16-15+7 'indb = @w[i], inda = @w[i-15+7]
         setinda --7      's0 = (w[i-15] -> 7) ^ (w[i-15] -> 18) ^ (w[i-15]
>> 3)      mov     indb,inda--
         mov    x,indb
         rol   x,#18-7
         xor   x,indb
         ror   x,#18
         shr   indb,#3
         xor   indb,x
         add   indb,inda    'w[i] = s0 + w[i-16]
         setinda ++14     's1 = (w[i-2] -> 17) ^ (w[i-2] -> 19) ^ (w[i-2]
>> 10)     mov     x,inda
         mov    y,x
         rol   y,#19-17
         xor   y,x
         ror   y,#19
         shr   x,#10
         xor   x,y
         add   indb,x      'w[i] = s0 + w[i-16] + s1
         setinda --5     'w[i] = s0 + w[i-16] + s1 + w[i-7]
:sch      add    indb++,inda

' Load variables from hash
         reps   #8,#1      'copy hash[0..7] into a..h
         setinds #a,#hash
         mov    indb++,inda++

' Do 64 hash iterations on variables
         reps   #64,@:itr  'i = 0..63
         setinds #k+0,#w+0 'indb = @k[i], inda = @w[i]

```

```

mov     x,g           'ch = (e & f) ^ (le & g)
xor     x,f
and     x,e
xor     x,g

mov     y,e           's1 = (e -> 6) ^ (e -> 11) ^ (e -> 25)
rol     y,#11-6
xor     y,e
rol     y,#25-11
xor     y,e
ror     y,#25

add     x,y           't1 = ch + s1
add     x,indb++      't1 = ch + s1 + k[i]
add     x,inda++      't1 = ch + s1 + k[i] + w[i]
add     x,h           't1 = ch + s1 + k[i] + w[i] + h

mov     y,c           'maj = (a & b) ^ (b & c) ^ (c & a)
and     y,b
or      y,a
mov     h,c
or      h,b
and     y,h

mov     h,a           's0 = (a -> 2) ^ (a -> 13) ^ (a -> 22)
rol     h,#13-2
xor     h,a
rol     h,#22-13
xor     h,a
ror     h,#22

add     y,h           't2 = maj + s0

mov     h,g           'h = g
mov     g,f           'g = f
mov     f,e           'f = e
mov     e,d           'e = d
mov     d,c           'd = c
mov     c,b           'c = b
mov     b,a           'b = a

add     e,x           'e = e + t1

:itr    mov     a,x           'a = t1 + t2
        add     a,y

' Add variables back into hash

        reps   #8,#1       'add a..h into hash[0..7]
        setindb #hash,#a
        add     indb++,inda++

hash_block_ret  ret
,
,
' Defined data
,
zero          long    0
d0            long    1 << 9

opad          long    $36363636 ^ $5C5C5C5C

hash_init     long    $6A09E667, $BB67AE85, $3C6EF372, $A54FF53A, $510E527F, $9B05688C,
$1F83D9AB, $5BE0CD19 'fractionals of square roots of primes 2..19

k             long    $428A2F98, $71374491, $B5C0FBCF, $E9B5DBA5, $3956C25B, $59F111F1,
$923F82A4, $AB1C5ED5 'fractionals of cube roots of primes 2..311
             long    $D807AA98, $12835B01, $243185BE, $550C7DC3, $72BE5D74, $80DEB1FE,
$9BDC06A7, $C19BF174
             long    $E49B69C1, $EFBE4786, $0FC19DC6, $240CA1CC, $2DE92C6F, $4A7484AA,
$5CB0A9DC, $76F988DA
             long    $983E5152, $A831C66D, $B00327C8, $BF597FC7, $C6E00BF3, $D5A79147,
$06CA6351, $14292967
             long    $27B70A85, $2E1B2138, $4D2C6DFC, $53380D13, $650A7354, $766A0ABB,
$81C2C92E, $92722C85
             long    $A2BFE8A1, $A81A664B, $C24B8B70, $C76C51A3, $D192E819, $D6990624,
$F40E3585, $106AA070
             long    $19A4C116, $1E376C08, $2748774C, $34B0BCB5, $391C0CB3, $4ED8AA4A,
$5B9CCA4F, $682E6FF3

```

```

long $748F82EE, $78A5636F, $84C87814, $8CC70208, $90BEFFFA, $A4506CEB,
$BEF9A3F7, $C67178F2
,
, Undefined data
,
hmac      res    1
bytes     res    1
count     res    1
length    res    1

opad_key  res    16

hash      res    8
hashx     res    8

w         res    64

a         res    1
b         res    1
c         res    1
d         res    1
e         res    1
f         res    1
g         res    1
h         res    1

x         res    1
y         res    1

```

ROM_Monitor

Here is the Propeller 2 monitor program listing. It runs when no other boot method is detected and it communicates serially, using an auto-baud detection routine triggered by an ASCII space character received. Additionally, the monitor is designed to be callable by your program with serial communications happening either on the default pins, or pins you specify.

Propeller ROM ends at **\$0E7F**, marked with the string "**== End of ROM ==**", clearly visible when using the monitor to view low HUB memory addresses:

```
>0e60.0f00
00E60- 0D 28 04 00 2D 80 04 00 89 00 00 00 8B 0A C6 96      '.(.-.....'
00E70- 3D 3D 20 45 6E 64 20 6F 66 20 52 4F 4D 20 3D 3D      '== End of ROM =='
00E80- 47 FE C1 0D B2 6A 7C 0C C1 46 7C 08 C1 48 7C 08      'G...j...F...H|.'
00E90- C1 4A 7C 08 C1 4C 7C 08 C1 4E 7C 08 C1 50 7C 08      '.J|..L|..N|..P|.'
00EA0- C1 52 7C 08 C1 54 7C 08 B2 6A 7C 0C 35 56 BC A0      '.R|..T|..j|.5V..'
00EB0- 00 00 00 00 E2 20 FC 0D 47 FE C1 0D 0D 5A FC 0C      '.....G...Z..'
00EC0- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      '.....'
00ED0- B1 82 FD 0C 00 00 00 00 00 00 00 00 00 00 00 00      '.....'
00EE0- 0D 5C FC 0C 2E 5E BC A0 2D 5E BC 84 36 60 3C 08      '\...^.-^..6<..'
00EF0- 37 62 3C 08 38 64 3C 08 39 66 3C 08 3A 5E 3C 08      '7b<.8d<.9F<.:^<.'
```

The first user writable location is **\$0E80**, which is the beginning of RAM.

Hardware multi-tasking is used to run concurrent auto-baud detection, serial input, and main monitor code:

```
'*****
'*
'*          Propeller II ROM Monitor          *
'*
'*          Version 0.1                       *
'*
'*          11/01/2012                        *
'*
'******
,
, usage:      cognew($70C, tx_pin << 9 + rx_pin)      'start monitor in new cog
,

CON

branch1_      = 0
branch2_      = branch1_      + 31
branch3_      = branch2_      + 35
hello_        = branch3_      + 15
error_        = hello_        + 33
hitspace_     = error_        + 11
spquote_      = hitspace_     + 10
quotechr_     = spquote_      + 4
sub0_         = quotechr_     + 3          'must be => $80
sub1_         = sub0_         + 4
sub2_         = sub1_         + 12
sub3_         = sub2_         + 12
help_         = sub3_         + 4

DAT

'*****
'* Data *
'*****

branch1       byte    cmd_new,      0
               byte    cmd_byte,    "Y"
               byte    cmd_word,    "W"
               byte    cmd_long,    "N"
               byte    cmd_viewp,   ". "
               byte    cmd_search,  "/"
```

```

byte cmd_enter, ":"
byte cmd_map, "M"
byte cmd_clrp, "L"
byte cmd_setp, "H"
byte cmd_notp, "T"
byte cmd_offp, "Z"
byte cmd_getp, "R"
byte cmd_quit, "Q"
byte cmd_help, "?"
byte 0 '31 bytes

branch2 byte cmd_view2, 0
byte cmd_view2, " "
byte cmd_range, "."
byte cmd_search2, "/"
byte cmd_enter2, ":"
byte cmd_watch, "@"
byte cmd_clkset, "*"
byte cmd_coginit, "+"
byte cmd_cogstop, "-"
byte cmd_clrp, "L"
byte cmd_setp, "H"
byte cmd_notp, "T"
byte cmd_offp, "Z"
byte cmd_getp, "R"
byte cmd_watchp, "#"
byte cmd_cfgp, "|"
byte cmd_setdacs, "\"
byte 0 '35 bytes

branch3 byte cmd_view3, 0
byte cmd_view3, " "
byte cmd_search3, "/"
byte cmd_enter3, ":"
byte cmd_move, ">"
byte cmd_move, "<"
byte cmd_checksum, "^"
byte 0 '15 bytes

hello byte 13,13,"=== Propeller II Monitor ===",13,13
byte 0 '33 bytes

error byte "? - Help"
byte 13,7,0 '11 bytes

hitspace byte "Hit SPACE",0 '10 bytes

spquote byte " '" ,0 '4 bytes

quotecr byte """,13,0 '3 bytes

sub0 byte " -",13,$80 '4 bytes
sub1 byte "{adr{.adr}}", $80 '12 bytes
sub2 byte "{dat{ dat}}", $80 '12 bytes
sub3 byte "adr", $80 '4 bytes

help byte 13, "~HUB", sub0_
byte sub1_, "~view", 13
byte sub1_, "/", sub2_, "`Search", 13
byte sub1_, ":", sub2_, "`Enter", 13
byte sub3_, ".", sub3_, "[</>]", sub3_, "`Move", 13
byte sub3_, ":", sub3_, "^", "`Checksum", 13
byte sub3_, "@", "`Watch", 13
byte "[Y/W/N]", "`Byte/word/long", 13
byte "~COGS", sub0_
byte "cog+", sub3_, "{+", sub3_, "}", "`Start", 13
byte "cog-", "`Stop", 13
byte "M", "`Map", 13
byte "~PINS", sub0_
byte "{pin}[H/L/T/Z/R]", "`High/low/toggle/off/read", 13
byte "pin#", "`Watch", 13
byte "pin|cfg", "`Configure", 13
byte "dat\", "`Set DACs", 13
byte "~MISC", sub0_
byte "dat*", "`Set clock", 13
byte "", "`Repeat", 13
byte "Q", "`Quit", 13 '(0 long follows)

longs long

```

```

'*****
'* Entry *
'*****

                                org
entry                            long    0                                'start of data string = 0/nop

                                reps    #1F6-reserves,#1          'clear reserves
                                setinda #reserves
                                mov     inda+,#0

                                getptr  rx_pin                    'get rx/tx pins
                                getptr  tx_pin
                                shr     tx_pin,#9
                                setp    tx_pin

                                getptrb base                        'get base address of byte data
                                sub     base,#1longs<<2

                                jmptask #baud_task,#%0010          'enable baud detector task
                                settask %%0101

                                tjz     period,#$                  'wait for <space> to set period

                                jmptask #rx_task,#%0100           'enable serial receiver task
                                settask %%0121

                                mov     wsize,#1                    'init word size to byte
                                call    #set_size

                                pusha  #0                          'init input line to <enter>
                                setptr  #hello_                    'print hello message

dmax                              'end of data string

'*****
'* Main Task *
'*****

message                            call    #tx_string          'print hello/error message
cmd_new                             call    #rx_line           'get input line
                                if_z   call    #parse              'parse first term
                                tjz    x,#cmd_view1              'if no hex and eol, view data
                                jmp    #cmd_go                    'else, process command

cmd_next_cr1f                       call    #tx_cr1f          'print cr/lf
cmd_next                             addspa  #1                  'skip chr

cmd_loop                             call    #parse              'parse next term
cmd_go if_nz                          jmp    #cmd_hex            'if hex, branch
                                movd  pinx,#z                    'pin update redirected to z
                                setptr #branch1_                 'not hex, vector by chr
                                call   #vector                    'if returns, no match

cmd_error                            setptr  #error_           'print error message
                                jmp    #message

cmd_hex                              mov     v1,value          'hex, save v1
                                movd  pinx,#pin                  'pin update okay
                                setptr #branch2_                 'vector by chr
                                call   #vector                    'if returns, no match
                                jmp    #cmd_view2                'view data

cmd_range                             call    #parse_next        'hex., get hex
                                if_z   jmp    #cmd_viewp2         'if no hex, view data

                                mov     v2,value                  'hex.hex, save v2
                                setptr #branch3_                 'vector by chr
                                call   #vector                    'if returns, no match
                                jmp    #cmd_view3                'view data
,
,
' Byte/word/long data

```



```

                djnz    z,#:loop          'loop until enter done
                getptrb enter              'update pointer
                jmp     #cmd_loop          'next command
,
,
,
; Move data
cmd_move       mov     y,x                'save ">"/"<"
words         call    #check_range        'check 1st address range, get number of
                call    #parse_hex         'get 2nd address
                max     value,amask        'v1=1st, value=2nd, z=words, y=">"/"<"
                and     value,amask
                cmp     y,#"<"            wz   'if "<", swap v1 and value
if_z          mov     x,v1
if_z          mov     v1,value
if_z          mov     value,x            'v1=from, value=to, z=words
                cmp     v1,value          wc   'if from < to, downward move
if_c          mov     x,z
if_c          shl     x,shift
if_c          add     v1,x
if_c          add     value,x
if_c          xor     rdxxxx,##%001_11110 'modify 'rdxxxx value,--ptrb'
if_c          xor     wrxxxx,##%001_11110 'modify 'wrxxxx value,--ptrb'
                setptrb v1
                setptrb value            'set pointers
:loop         call    #rdxxxx
                call    #wrxxxx
                djnz    z,#:loop          'move data
                xor     rdxxxx,##%001_11110 'restore 'rdxxxx value,ptrb++'
if_c          xor     wrxxxx,##%001_11110 'restore 'wrxxxx value,ptrb++'
                jmp     #cmd_loop          'next command
,
,
,
; checksum
cmd_checksum   call    #check_range        'check range
:loop         setptrb v1
                call    #rdxxxx
                add     y,value
                djnz    z,#:loop
                mov     value,y
                mov     hsize,#8
                call    #tx_hex
                jmp     #cmd_next_cr1f     'next command
,
,
,
; watch
cmd_watchcp   movs    rdxxxj,#rdxxx_ret  wz   'set pin mode, z=0
                mov     hsize,#1
cmd_watch     if_z    movs    rdxxxj,#rdxxxm
if_z          mov     hsize,wsz
if_z          shl     hsize,#1
                call    #rdxxxp           'get initial value
:loop         mov     z,value              'preserve value
                call    #tx_hex
                call    #tx_space          'print value
                'print space
:wait        if_nz    call    #rx_check
if_nz          jmp     #cmd_next_cr1f     'if key hit, exit
                call    #rdxxxp           'get current value

```

```

        if_z    cmp     value,z      wz      'if same, check again
               jmp     #:wait
               jmp     #:loop        'new value, loop
,
,
,
: clkset
cmd_clkset   setptr  #hitspace_
             call   #tx_string     'print hit-space message
             clkset v1             'set clk
:wait       call   #rx              'wait for space
             cmp    x,#" "         wz
             if_nz  jmp     #:wait
             jmp     #cmd_next_crlf 'next command
,
,
,
: coginit
cmd_coginit  setcog  v1            'set cog
             call   #parse_hex     'get program address
             mov    y,value        'save program address
             mov    value,#0       'clear pointer address
             if_z   cmp    x,#"+"   wz      'if '+', get pointer address
             call   #parse_hex
             coginit y,value       'do 'coginit program,pointer'
             jmp    #cmd_loop      'next command
,
,
,
: Cogstop
: Quit
cmd_quit    cogid  v1            'quit
cmd_cogstop cogstop v1          'stop cog
             jmp    #cmd_next     'next command
,
,
,
: Map
cmd_map     mov    y,#7          'ready for 7..0
cmd_map_loop call   #tx_space     'print space
             mov    x,y           'get cog status
             cogid  x              wc
cmd_map_c   cmp    x,y           wz
             if_nc  mov    x,#"0"   'get 0/1/M chr
             if_c   mov    x,#"1"
             if_z   mov    x,#"M"
             call   #tx            'print chr
             sub    y,#1          wc
             if_nc  jmp     #cmd_map_loop 'loop until done
             jmp    #cmd_next_crlf 'next command
,
,
,
: Pin writes clrp/setp/notp/offp
: Pin read
cmd_clrp    movs   pinop,$DA     wz      'clrp, z=0
cmd_setp   if_z   movs   pinop,$DB wz      'setp, z=0
cmd_notp   if_z   movs   pinop,$D9 wz      'notp, z=0
cmd_offp   if_z   movs   pinop,$D8 wz      'offp, z=0
cmd_getp   if_z   movs   pinop,$D6 wz      'getp, z=1
pinx       mov    pin,v1         'if hex, get pin (d = pin/z)
pinop      getp   pin            wc      'becomes clrp/setp/notp/offp/getp
             if_z   jmp     #cmd_map_c 'if getp, show pin value

```

```

        jmp      #cmd_next          'next command
    ;
    ; Pin configuration
cmd_cfgp      call      #parse_hex    'get configuration
                setport v1          'set pin port
                decod5  v1          'get pin mask
                cfgpins v1,value     'configure pin
        jmp      #cmd_loop          'next command
    ;
    ; Setdacs
cmd_setdacs   setdacs v1            'set all four dacs with 8-bit values
        jmp      #cmd_next          'next command
    ;
    ; Help
cmd_help      setptr  #help_         'print help message
                call   #tx_string
        jmp      #cmd_next_cr1f     'next command

'*****
'* Main Task Subroutines *
'*****
    ;
    ; Vector branch
vector        addptr  base           'add data base pointer
vector_loop   rbyte   z,ptr++        'get jump address
vector_ret    tzj     z,#0           'if 0, no match found, return
                rbyte   y,ptr++        'get target
                xor     y,x           'compare to x
                if_nz   jmp     #vector_loop    wz    'if no match, loop
        jmp      z                   'match found, jump, y=0, z=1
    ;
    ; Check address range (v1..v2)
check_range   max     v1,amask       'trim v1
                and    v1,amask
                max     v2,amask       'trim v2
                and    v2,amask
                if_c    cmp     v2,v1   wc    'make sure v2 => v1
                jmp     #cmd_error
                mov     z,v2           'get number of words
                sub    z,v1
                shr    z,shift
                add    z,#1
check_range_ret ret
    ;
    ; Set rdxxxx/wrxxxx and others by word size
set_size      test    wsize,#%010    wc    'set rdxxxx/wrxxxx by word size
                setbc  rdxxxx,#26
                setbc  wrxxxx,#26
                test    wsize,#%100   wc
                setbc  rdxxxx,#27
                setbc  wrxxxx,#27
                mov     shift,wsize   'set shift by word size
                shr    shift,#1
                mov     amask,wsize   'set amask by word size

```

```

sub    amask,#1
xor    amask,h0001FFFF

and    view,amask
and    enter,amask

set_size_ret    ret

rdxxxp    if_nc    getp    v1
if_c      mov     value,#0    wc    'read pin as "0" or "1"
rdxxvj    mov     value,#1
rdxxxm    jmp     #rdxxx_ret    'd = rdxxx_ret/rdxxxm

rdxxxm    setptr    v1    'read mem

rdxxxx    rdbyte   value,ptr++    'rdbyte/rdword/rdlong
rdxxxp_ret    ret
rdxxxx_ret

wrxxxx    wrbyte   value,ptrb++    'wrbyte/wrword/wrlong
wrxxxx_ret    ret

;
; Input line
;
rx_line    setspa   #0    'point to start of line

mov     x,#">"
call    #tx    'show prompt

call    #rx
if_nz   cmp     x,#""    wz    'get first chr
        jmp     #:first    'check for repeat
        'if not repeat, first chr

:show     popar    x    wz    'repeat, show line
if_nz   call    #tx
if_nz   jmp     #:show
        jmp     #:done

:loop     call    #rx    'get next chr

:first    if_z     cmp     x,#13    wz    'cr?
        jmp     #:cr

        if_nz   cmp     x,#8    wz    'backspace?
if_nz   cmp     x,#127    wz
if_z     jmp     #:bs

        if_nc   cmp     x,#" "    wc    'visible chr?
if_c     cmp     x,#"~"    wc
if_c     jmp     #:loop

        if_c     pusha   x    wc    'visible chr, append to line
if_c     chkspa  #1    wc    'overflow?
if_nc   subspa  #1    'if overflow, back up
        call    #tx    'if not overflow, print chr
        jmp     #:loop

:bs       if_nz   chkspa  #1    wz    'backspace, line empty?
if_nz   pushar  x    'if not empty,
if_nz   call    #tx    '..print backspace
if_nz   call    #tx_space    '..print space
if_nz   popar  x    '..print backspace
if_nz   call    #tx
if_nz   subspa  #1    '..back up
if_nz   jmp     #:loop

:cr       pusha  #0    'cr, end line with 0

:done     setspa  #0    'point to start of line

tx_cr1f   mov     x,cr1f    'print cr/lf
call     #tx

tx_cr1f_ret    ret
rx_line_ret    ret
;

```

```

' Parse hex/text data
,
parse_data      mov     w,#0          'reset data count
                setinda #0        'point to string data

:hex            call    #parse_next  'hex loop, check hex
                call    #enter_data  'if hex, enter value
                cmp     x,#" "      wz   'check for space (more hex)
                if_z    jmp     #:hex  'if more hex, loop

                cmp     x,#"'"      wz   'not hex, "'"?
                if_nz   jmp     #:done 'if not "'", done

:text           addspa  #2          'text loop
                popa   x           'get and point to next chr
                cmp     x,#"'"      wz   'check for "'"
                if_z    jmp     #:hex  'if "'", back to hex
                tjz    x,#:done      'if eol, done
                mov     value,x      'text chr
                call    #enter_data  'enter chr
                jmp     #:text       'loop

:done          sub     w,#1          'get data count
                if_nc   mov     dsize,w 'if 0, reuse old data

:fix           movd   :fix,dsize    'form circular buffer
with 1:4 threading)  fixinda #0,#0  '(no instruction-modification problem)
parse_data_ret  ret

enter_data     incmod  w,#dmax       'check if data limit exceeded
if_c          jmp     #cmd_error     'if data limit exceeded, error
                mov     inda++,value 'store value in data

enter_data_ret  ret
,
' Parse hex
,
parse_hex     call    #parse_next    'try to parse hex
if_z         jmp     #cmd_error      'if no hex, error

parse_hex_ret  ret
,
' Parse line (@spa), z=0 if hex (value)
,
parse_next    addspa  #1            'advance to next chr

parse        mov     value,#0        'z=1
                call    #skip_spaces  'skip any spaces (preserve z)

:loop        popar   x              'get chr
                call    #check_hex    'check hex
                shl     value,#4      'if hex, append nibble and loop
if_c        or     value,x
if_c        jmp     #:loop          wz   'z=0

                subspa  #1          'repoint to non-hex chr
                call    #skip_spaces  'skip any post-hex spaces (preserve z)

                if_c    call    #check_hex
                popa   x            'check hex
                if_c    jmp     #:loop 'if hex, back up to space chr

                if_c    cmpr   x,#"a"-1  'make non-hex chr uppercase
                if_c    cmp    x,#"z"+1
                if_c    sub    x,#"a"-"A"
parse_next_ret  ret
parse_ret     ret
,
' Skip spaces (@spa)
,
skip_spaces   popar   x             'skip space chr(s)
                cmp     x,#" "      wz
                if_z    jmp     #skip_spaces

```

```

                subspa #1                                'back up to non-space chr
skip_spaces_ret ret                                    wz      'restore z
,
, Check hex (x), c=1 if hex (x)
check_hex      cmpr   x,#"0"-1                          wc      '"0".."9" -> $0..$9
               if_c   cmp    x,#"9"+1                    wc
               if_c   add    x,#"A"-9"-1
               if_nc  cmpr   x,#"A"-1                          wc      '"A".."F" -> $A..$F
               if_c   cmp    x,#"F"+1                    wc
               if_c   add    x,#"a"-A"
               if_nc  cmpr   x,#"a"-1                          wc      '"a".."f" -> $A..$F
               if_c   cmp    x,#"f"+1                    wc
               if_c   sub    x,#"a"-10
check_hex_ret  ret
,
, Print range (v1..v2)
tx_range1     mov    v1,view                               'view.view + v2
tx_range2     and    v2,amask                             'v1..v1 + v2
               add    v2,v1
tx_range      call   #check_range                       'check range
               mov    view,v1                          'set address
:line         mov    value,view                          'print 5-digit address
               mov    hsize,#5
               call   #tx_hex
               call   #tx_dspace                       'print "- "
               mov    x,wsiz                             'get number of words on line
               rev    x,#32-5
               mov    v1,z
               max    v1,x
               mov    v2,v1                             'get number of ascii bytes on line
               shl    v2,shift
               sub    z,v1                               'update number of words left
:hex          setptr  view                               'print hex words
               call   #rdxxxx
               mov    hsize,wsiz
               shl    hsize,#1
               call   #tx_hex
               call   #tx_space
               djnz   v1,#:hex
               setptr  #spquote_
               call   #tx_string                       'print " "'
:ascii        setptr  view                               'print ascii bytes
               rdbyte x,ptr++
               cmp    x,#" "                            wc      'visible chr?
               if_nc  cmpr   x,#"~"                    wc
               if_c   mov    x,#"."
               call   #tx
               djnz   v2,#:ascii                       'substitute "." for non-visible chrs
               getptr  view                               'update address
               setptr  #quotecr_
               call   #tx_string                       'print "" + cr
               call   #rx_check
               tjnz   z,#:line                          'check key hit
               if_z   'if no key hit and more words left, print
another_line  tx_range1_ret
tx_range1_ret tx_range2_ret
tx_range2_ret tx_range_ret
tx_range_ret ret
,
, Print string (@ptr)

```

```

'
tx_string      addptrb base          'add data base pointer
tx_string_loop rdbYTE  x,ptrA++      'get chr
tx_string_ret  tjz     x,#0          'if 0, done
                test    x,$80          wZ      'substring?
                if_nz   notb tx_string_loop,#8  'toggle ptrA/ptrB
                if_nz   setptrb x          'ptrB points to substring
                if_nz   addptrb base
                if_nz   jmp     #tx_string_loop 'start substring or resume string
                cmp     x,#`"          wZ      'long tab?
                if_z    subR  y,#32-16
                if_nz   cmp    x,#"~"        wZ      'short tab?
                if_z    add    y,#16
:tab           if_z    call   #tx_space
                if_z    djnz  y,#:tab
                if_z    call   #tx_dspace
                if_z    jmp   #tx_string_loop
                if_z    cmp    x,#13          wZ      'cr?
                if_z    call   #tx_cr1f
                if_z    mov    y,#0
                if_nz   call   #tx
                if_nz   add    y,#1
                if_nz   jmp   #tx_string_loop
'
' Print hex (value)
tx_hex         mov     y,hsize
                shl     y,#2
                ror     value,y
                'pre-rotate to get 1st nibble in top
:loop         mov     y,hsize
                rol     value,#4
                mov     x,value
                call   #tx_nib
                djnz  y,#:loop
                'print nibbles
tx_hex_ret    ret
'
' Print "- "
tx_dspace     mov     x,dspace
                jmp     #tx
'
' Print space
tx_space      mov     x,#" "
                jmp     #tx
'
' Print nibble (x)
tx_nib        and     x,#$F
                'isolate nibble
                if_c    cmp    x,$A          wC      'alpha or numeric?
                if_nc   add    x,#"0"        'numeric
                if_nc   add    x,#"A"-$A    'alpha
'
' Transmit chr (x)
tx            shl     x,#1
                setb   x,#9
                'insert start bit
                'set stop bit
                getcnt w
                'get initial time
:loop         add     w,period
                passcnt w
                shr    x,#1          wC      'add bit period to time
                setpc tx_pin        'loop until bit period elapsed
                tjnz  x,#:loop      'get next bit into c
                'write c to tx pin
                'loop until bits done
tx_dspace_ret
tx_space_ret

```

```

tx_nib_ret
tx_ret      ret
'
' Receive chr (x)
rx          if_z    call    #rx_check      'wait for rx chr
            jmp     #rx
rx_ret      ret
'
' Check receiver, z=0 if chr (x)
rx_check    or      rx_tail,$80      'if start or rollover, reset tail
            if_nz   getspb  rx_temp    wz      'if head uninitialized, z=1
            cmp     rx_temp,rx_tail wz      'if head-tail mismatch, byte ready, z=0
            if_nz   getspa  rx_temp
            if_nz   setspa  rx_tail      'preserve spa
            if_nz   popar   x           'get tail
            if_nz   getspa  rx_tail      'get byte at tail
            if_nz   setspa  rx_temp     'update tail
            if_nz
            if_nz   setspa  rx_temp     'restore spa
rx_check_ret  ret

'*****
'* Serial Receiver Task *
'*****

rx_task     if_z    chkspb          wz      'if start or rollover, reset head
            setspb  #80
            mov     rx_bits,#9        'ready for 8 data bits + 1 stop bit
            neg     rx_time,period    'get -0.5 period
            sar     rx_time,#1
            jp      rx_pin,#$        'wait for start bit
period delay
            subcnt rx_time          'get time + 0.5 period for initial 1.5
:bit        rcr     rx_data,#1        'rotate c into byte
            add     rx_time,period    'add 1 period
            passcnt rx_time          'wait for center of next bit
            getp   rx_pin            wc      'read rx pin into c
            djnz   rx_bits,#:bit     'loop until 8 data bits + 1 stop bit
received
            shr    rx_data,#32-8     'align byte
            pushb  rx_data           'store byte at head, inc head
            jmp    #rx_task          'wait for next byte

'*****
'* Baud Detector Task *
'*****

baud_task   movd   ctr,rx_pin        'set ctra to time rx pin states
:loop       notb   ctr,#5            wc      'if 1,0 sample set, c=0
            setctra ctr              '($20 -> 10000001001 -> 1, 6x 0, 1x 1, 2x
0, 1)
            if_nc  mov   limh,buff0   'if 1,0 sample set,
            if_nc  shr   limh,#4      '..make window from 1st 0 (6x if $20)
            if_nc  neg   liml,limh
            if_nc  add   limh,buff0
            if_nc  add   liml,buff0
            if_nc  mov   comp,buff1   'if 1,0 sample set,
            if_nc  mul   comp,#6      '..normalize 2nd 1 (1x if $20) to 6x
            if_nc  cmpr  comp,limh    wc      '..check if within window
            if_nc  cmp   comp,liml    wc
            if_nc  mov   comp,buff2   'if 1,0 sample set,
            if_nc  mul   comp,#3      '..normalize 2nd 0 (2x if $20) to 6x

```



```

        if_nc    cmp    comp,limh    wc    '..check if within window
        if_nc    cmp    comp,liml    wc
        if_nc    add    buff0,buff2    'if $20,
        if_nc    shr    buff0,#3    '..compute period from 6x 0 and 2x 0
        if_nc    mov    period,buff0    '..update period

        mov    buff0,buff1    'scroll sample buffer
        mov    buff1,buff2

:wait    getcosa    buff2    'wait for next sample
        tjnz    buff2,#:loop
        jmp     #:wait

'*****
'* Constants *
'*****

h0001FFFF    long    $0001FFFF    'memory limit
crlf         long    1<<18 + $0A<<10 + $0D    'cr/lf
dspace       long    1<<18 + " "<<10 + "-"    'dash/space
ctr          long    %100_01001    'ctr configuration for timing low on rx
pin

'*****
'* variables *
'*****

reserves

rx_pin       res    1    'main task
tx_pin       res    1
base         res    1
w            res    1
x            res    1
y            res    1
z            res    1
v1           res    1
v2           res    1
value        res    1
view         res    1
enter        res    1
pin          res    1
dsize        res    1
hsize        res    1
wsize        res    1
shift        res    1
amask        res    1

rx_tail      res    1    'serial receiver task
rx_temp      res    1
rx_time      res    1
rx_data      res    1
rx_bits      res    1

buff0        res    1    'baud detector task
buff1        res    1
buff2        res    1
limh         res    1
liml         res    1
comp         res    1
period       res    1

```

Appendix B PASM Program Listings and Object Code

DE2-Counter-To-LED-Blinker

This program writes some of the slower changing global counter digits to the I/O pins connected to the onboard LED's. This program also writes a value to the hub and starts an instance of the monitor for the purpose of demonstrating how to watch addresses.

```
DAT
    org
start_mon    setcog #3          '+ %1000 --uncomment '+ %1000' for next available cog
              coginit monitor_pgm, monitor_ptr 'function; otherwise,
              'start monitor on cog n
              neg    dirb, #1          'make the port B I/Os outputs
:loop        getcmt A              'fetch lower 32 bits of global counter
              shr    A, #24           'shift away the fast incrementing digits
              shl    A, #8
              mov    pinb, A          'write slow ones to LED's on DE2 board
              wrword A, write_address 'Put a value in the hub to watch
              jmp    #:loop           'keep doing it!

monitor_pgm  long    $70C            'ROM entry point for monitor
monitor_ptr  long    90<<9 + 91     'serial pins = RX<<9 + TX

write_address long    $2000

A            res     1              'storage
```

Load and run this program with Pnut.exe to see the LED's flash and interact with the monitor to watch address \$2001 increment as the counter does.

```
>2001@
21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
>
```

Running Multiple Monitors

Here is a short program that does exactly that on the DE2 emulation, using the default p90, p91 and optional p15, p13 to communicate with two instances of the monitor running on COGS 3 and 0.

```
DAT
    org
start_mon    setcog #3          'set target COG 3
              coginit monitor_pgm, monitor_ptr 'start monitor on target COG
              setcog #0          'set target COG 0
              coginit monitor_pgm, monitor_ptr1 'start monitor on target COG

monitor_pgm  long    $70C            'ROM entry point for monitor
monitor_ptr  long    15<<9 + 13     'Pins for one monitor
monitor_ptr1 long    90<<9 + 91     'Pins for the other one
```

This program is useful to quickly test both serial connections, if you are using the dual Prop Plug connection scenario mentioned earlier in the text.

It's worth noting that either monitor isn't aware of the other one when more than one is running at a time. The output of the cog map (m) command for both monitors show the other one as an active COG, not a monitor cog, normally designated with "M":

Output from monitor instance running on COG 3:

```
=== Propeller II Monitor ===
```

```
>m
0 0 0 0 M 0 0 1
>
```

Output from monitor instance running on COG 0

```
=== Propeller II Monitor ===
```

```
>m
0 0 0 0 1 0 0 M
>
```

Start Monitor From HUB RAM Memory

You can download a modified version of the ROM Monitor setup to both run from HUB memory at an address you specify, and start with only the monitor cog start command. The default is \$1000, editable in the program values.

A sample object file you can paste right into a running monitor and source code is provided in the zip package found here:

<http://forums.parallax.com/attachment.php?attachmentid=98300&d=1356951096>

This code is provided with the assumption you might want to patch the monitor to provide different functionality, or use it as part of some other serial communications. Here is the usage information and key values needed to build and run it at a given address.

```
' Modified to be started from monitor, or coginit with parameters ignored
' eg: [cog]+11B4
'
' For different HUB RAM address load, set rx_pin, tx_pin, base below. Base = [load
address + $1B4] = entry for COGINIT
```

```
rx_pin      long    15<<9 + 13    '15          'main task
tx_pin      long    15<<9 + 13    '13          'Pins --> ptrA
base        long    $11b4        'Entry Point --> ptrB
```

Note, the code is compiled at \$0, with the values setup for loading at \$1000, and the monitor entry point is that base address \$1000 + \$1B4, assuming no modifications are done to the text data starting at \$1000.

Replace_Example.spin

This is a simple program that blinks the LED's in different ways and on different COGS with the monitor running for "while running" modifications.

```
' Terasic DE2-115 Prop2 Counter To LED
' -----
DAT
      org
start_mon      setcog #3      '+ %1000      'uncomment '+ %1000' for
                                     'next available cog
```

```

        coginit monitor_pgm, monitor_ptr    'function; otherwise,
                                           'start monitor on cog n
        setcog #1000
        coginit entry_addr, A             'start simple blinker on a COG
        neg     dirb, #1                   'make the port B I/Os outputs

:loop
        getcnt A                          'fetch lower 32 bits of global counter
        shr   A, #24                       'shift away the fast incrementing digits
        shl   A, #8                         'write upper counter digits to LED's
        mov   pinb, A
        wrword A, write_address            'Put a value in the hub to watch with monitor
        jmp   #:loop                       'keep doing it!

monitor_pgm    long    $70C                'ROM entry point for monitor,
monitor_ptr    long    90<<9 + 91         'serial pins = RX<<9 + TX
write_address  long    $2000              'write truncated counter value to some HUB
address
entry_addr    long    @entry+$e80        'Define correct address for COGINIT
A             res     1                   'storage

' Terasic DE2-115 Prop2 Simple One Pin LED Blinker
' -----
DAT
        org                'new COG address origin
entry    setp              pin            'set specified pin to output
        shl                toggle_mask, pin 'setup to toggle pin state

loop     getcnt            Time           'fetch lower 32 bits of global counter
        add                Time, delay   'prepare to wait on future counter value
        add                Time, #5f     'a minimum delay
        WAITCNT            Time, Delay    'wait for it
        xor                pinb, toggle_mask 'toggle the pin
        rdlong             Delay, delay_address 'update delay value from HUB
        jmp                #loop         'keep doing it!

Pin      long              33             'define pin to blink
Delay    long              $1c9c380      'slow blink = 30_000_000 = 1/2 sec @ 60Mhz
Delay_address long        @Delay + $e80  'calculate correct HUB address for WAITCNT
toggle_mask long          1             'pin state mask

Time     res               1             'storage

```