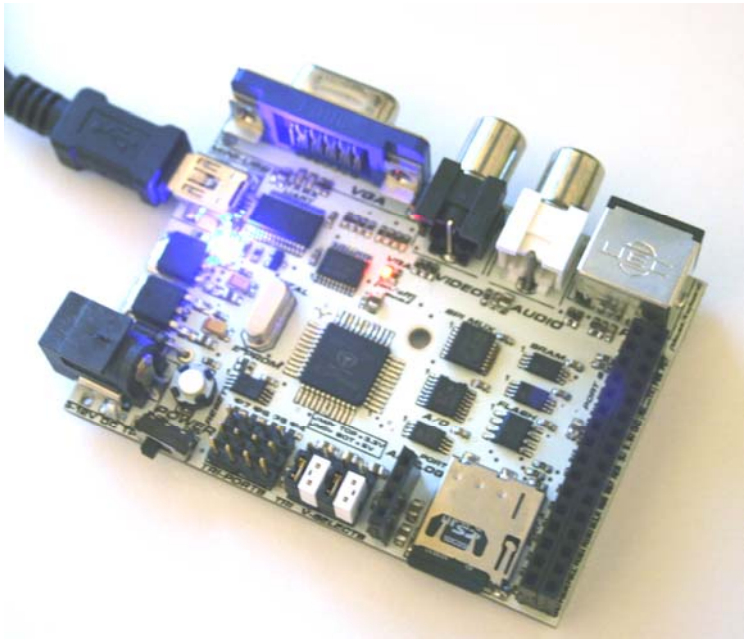
Web Site: www.parallax.com
Forums: forums.parallax.com
Sales: sales@parallax.com
Technical: support@parallax.com

Office: (916) 624-8333
Fax: (916) 624-8003
Sales: (888) 512-1024
Tech Support: (888) 997-8267

Unleashing the Propeller™ C3

by André LaMothe



Overview

Welcome to **"Unleashing the Propeller C3."** This hands-on guide covers the design and hardware of the C3 as well as numerous demos and tutorials to get you going fast. First and foremost the C3 is a Propeller-based product, so everything you know about the multicore Propeller microcontroller applies. If you're a seasoned expert then you might just want to skim this manual focusing on the SPI bus, pin outs, and peripherals since that's all you need to get started. However, if you're new to the Propeller chip then you will want to read this manual carefully as well as take a look at the numerous online and printed manuals covering the Propeller chip. Start here at the Parallax Propeller site to see what's available online:

www.parallax.com/propeller

Specifically be sure to review these documents especially:

- Propeller Manual (from the Propeller Downloads link)
- Propeller Datasheet (from the Proeller Downloads link)
- Propeller Questions & Answers system (from the Propeller Q&A link)

Also, there are a number of printed books about the Propeller you might want to pick up and read:

- *Programming and Customizing the Multicore Propeller Microcontroller*, Parallax.
- *Game Programming for the Propeller Powered HYDRA*, André LaMothe.
- *Programming the Propeller with Spin, A Beginners Guide to Parallel Processing*, Harprit Sandhu.

This manual does not teach you Spin, assembly language, graphics, or how to write drivers for the Propeller chip. This manual only covers the Propeller C3 hardware platform and software demos that come with it. If there is particular material that is beyond the scope of this manual, I will refer you to one of the books above in many cases. Of course, I hope you by default read the Propeller Reference Manual itself!

Table of Contents

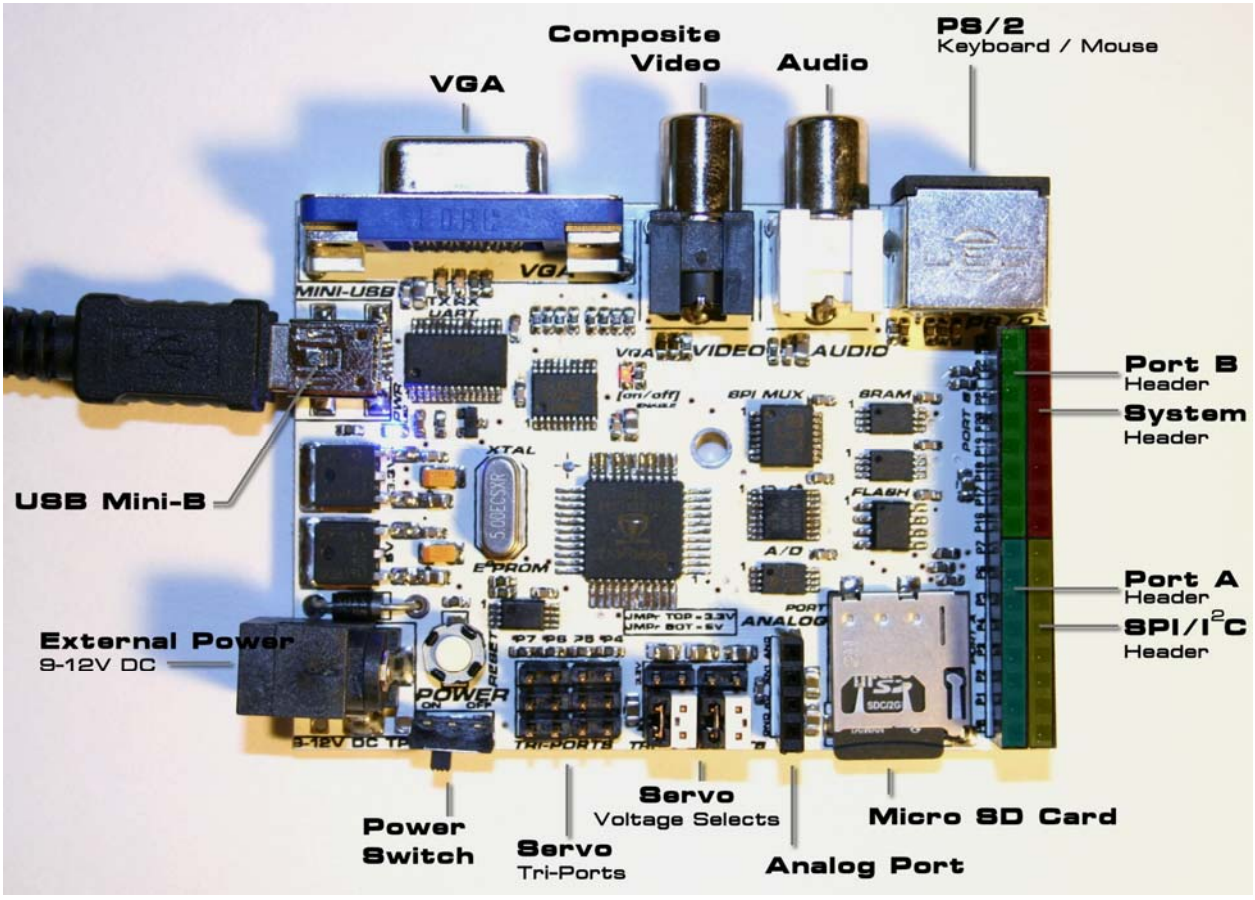
1 Quick-Start Guide

In this brief quick-start guide, we are going to review the Propeller C3 board briefly, plug it in, power it up, and download a demo to it. This will confirm everything is working properly as well as familiarize you with the C3 board itself.

First off, take a look at Figure 1.1 which is an annotated image of the C3. Take a moment to review all the interfaces and location of various chips. Later we will drill down closer and each of the IO headers will be illustrated, but for now, just get a birds-eye view of the board.

Now, that you have an idea of what goes where, let's go ahead and plug the board in and power it up! The Propeller C3 comes pre-loaded with a test suite that exercises all the hardware including the optional microSD card and NES adapter. Don't worry if you don't have these, the test will just ignore them or print out appropriate information.

Figure 1.1 — Annotated Propeller C3 Board



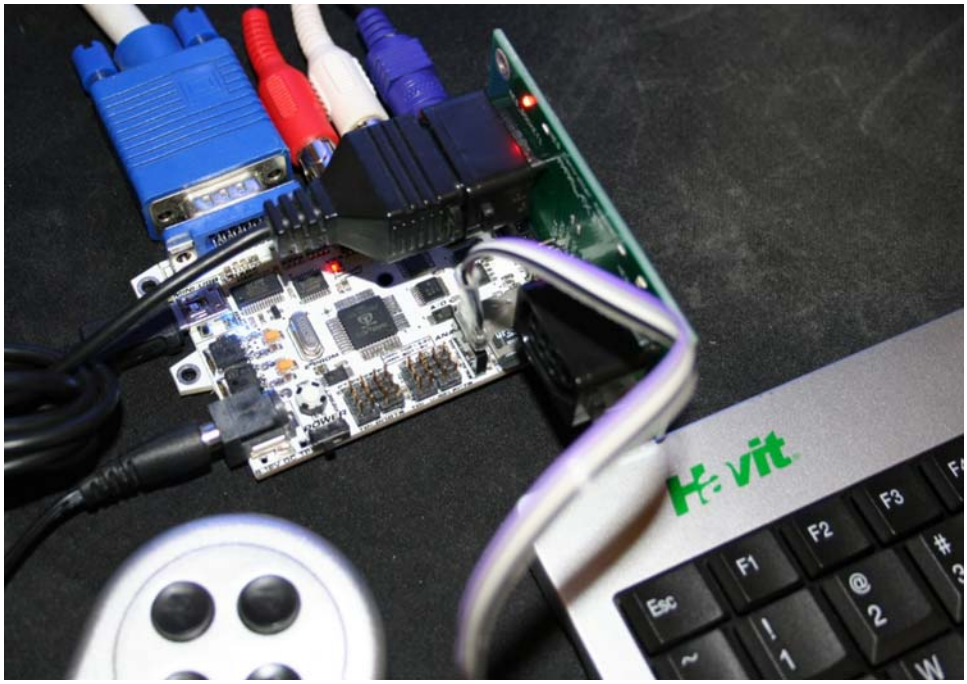
!	**The Propeller C3 Is designed to accept power from the USB port OR from an external power supply. Do not simultaneously supply power from more than one source. Before supplying external power be sure to remove the USB cable, otherwise damage to the C3 and your computer may occur.**

1.1 Demo Test: Power-Up with No PC

The demo test suite comes pre-loaded on the C3; it tests out the entire board including NTSC, VGA, PS/2 keyboard, microSD slot, A/D converter, sounds, status LED, and IO header. However, all you need is the C3 unit and an NTSC TV or VGA at minimum along with an external power supply. But, if you want to test everything out and hook everything up, you will need the following:

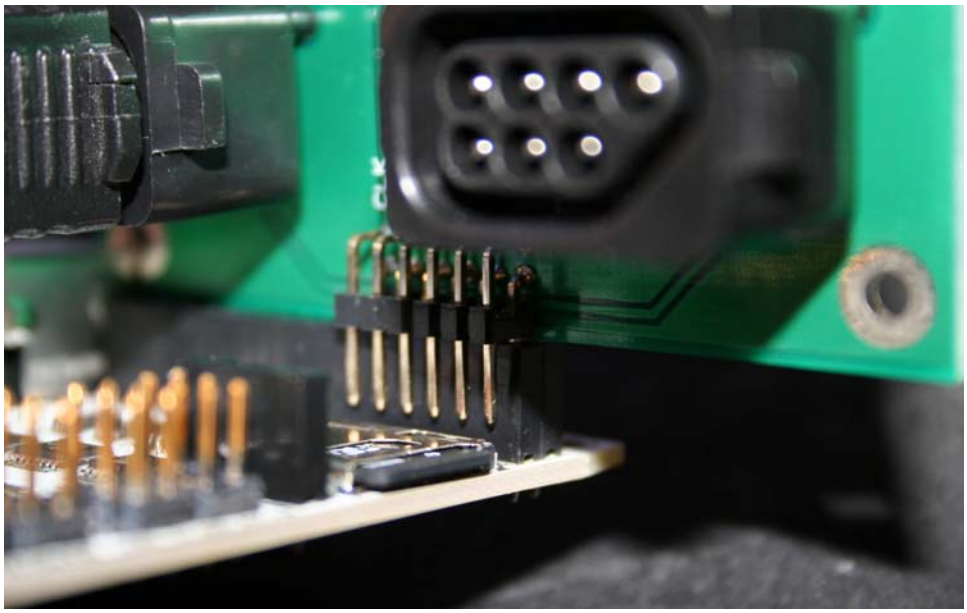
- NTSC TV/monitor
- VGA Monitor
- PS/2 Keyboard
- Parallax NES Gamepad Adapter (part #32368)
- A NES compatible gamepad
- MicroSD card formatted FAT16
- Simple potentiometer circuit to test 2-channel A/D converter.
- 7.5–9 VDC 300 mA, 2.1 mm ID, 5.5 mm OD, center positive power supply

Figure 1.2 — The Propeller C3 with everything hooked up.



Step 1. (Prepping Connections) — Make sure the power switch on the C3 is in the OFF position to the right. Now, hookup your C3 as shown in Figure 1.2, but WITHOUT the USB connection. If you're hooking up the A/D converter, NES gamepad adapter, and/or SD card, follow all the steps; otherwise skip to Step 5 on page 8.

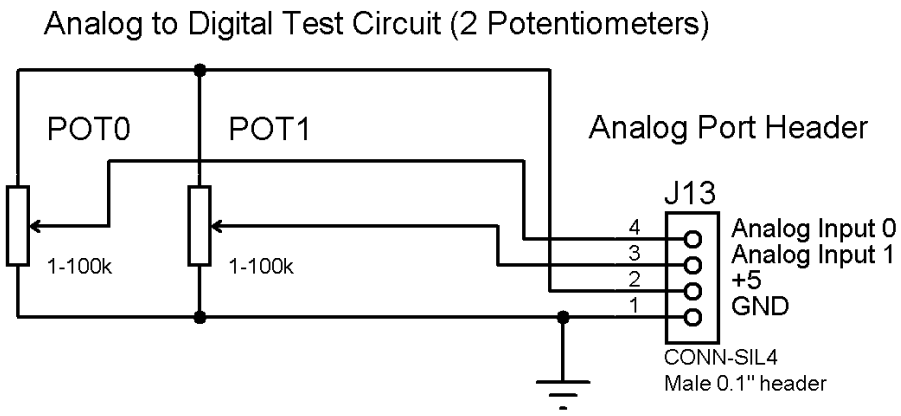
Figure 1.3 — Close-up of correct insertion of Parallax NES adapter into C3



Step 2. (NES Adapter / Optional) — If you have a Parallax NES Gamepad Adapter and NES-compatible gamepad then you can interface it to the C3. Later in the manual we will discuss how the interface works, but for now, simply insert the game pad adapter into the C3 main header as shown in Figure 1.3. Notice that only one row of the NES adapter is inserted into the main 2x16 header of the C3. The correct position is at the bottom/left of the 2x16 header as shown in the figure above.

Step 3. (A/D converter / Optional) — The C3 has a dedicated 2-channel 12-bit analog to digital converter (Microchip MCP3202) on the system SPI bus. The test suite displays the voltages on each channel. If you don't plug anything into the A/D channels they will pick up random noise; however, if you inject a signal into each channel then the test suite displays the values of each channel 0..5 V as a 0..4095 on the screen. A simple circuit to generate a variable voltage is shown in Figure 1.4. Basically, the A/D port labeled top to bottom is Analog 1, Analog 0, +5 V and ground. So, you can power a simple voltage divider or potentiometer circuit as shown in the figure. The 5 V is placed across two potentiometers (1 kΩ–100 kΩ will do), and the wiper of each is connected to the analog inputs. As you turn the potentiometers the voltage on the wiper will vary from 0 to 5 V and hence the value read by the MCP3202 will vary 0 to 4095.

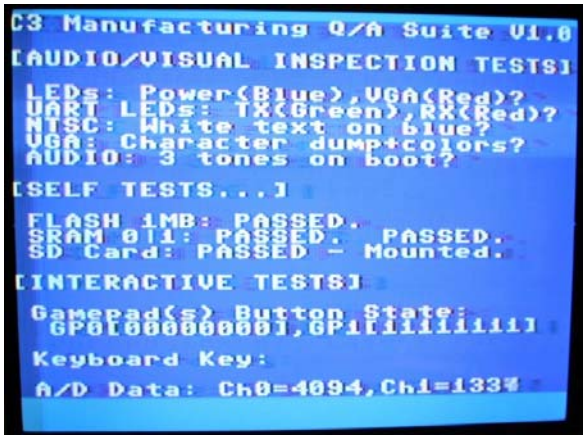
Figure 1.4 — A/D test circuit

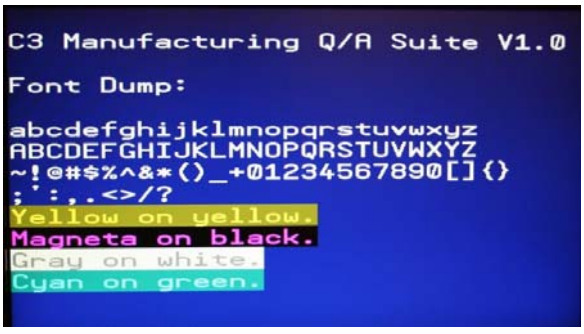


Step 4. (MicroSD card / Optional) — The C3 has an SPI-based microSD slot with spring-loaded insert and eject (push-push). If you have a microSD card, you can insert it and the test suite will mount the card and read the partition tables. The C3 will not write anything to the card. Note: You must use a FAT16 formatted SD card.

Figure 1.5 — C3 hardware boot screen

Figure 1.6 — C3 Test and Q/A suite running on NTSC screen (left) and on VGA Screen (right).





Step 5. (Run the test suite) — You should have your C3 all hooked up now, so its time to power it up and watch the test suite run. Slide the power switch to the ON position which is to the left. Depending on what you have hooked up to the C3, it will drive the VGA, NTSC, and all other peripherals. First, you will see a brief "C3 boot screen" as shown in Figure 1.5, then the main test suite will start running as shown in Figure 1.6 (left). Additionally, if you have a VGA monitor connected to the C3 you will see the image shown in Figure 1.6 (right) as well. However, this display is static and only verifies VGA functionality. Thus, you need an NTSC display to get the full use of the test suite. The test suite is composed of three primary tests:

- Audio/Visual Inspection Tests — These are simply tests you do yourself and verify.
- Self Tests — These tests are automated and run themselves.
- Interactive Tests — These require your input and extra peripherals.

The self tests test the onboard FLASH memory, SRAM banks, SD card (if inserted) and then fall through to the interactive tests. These tests require your input and display the status of the PS/2 keyboard (if plugged in), gamepad (if plugged in), and finally the A/D converter if you inject 0..5 V signals into either or both of the A/D channels.

That completes the power-up test. Next, let's move onto compiling and downloading the demo.

1.2 Demo Test: Compile & Download from PC

In this part of the Quick-Start we are going to use the Propeller Tool IDE to compile and download the test suite to the C3. If you're already proficient at using the Propeller Tool then you can skip this section, but if you're a new user then you should follow along.

This manual doesn't teach Propeller programming, Spin, or using the Propeller Tool. For those subject areas, please refer to the resources outlined in the Overview section above, which lists online and printed manuals you should read.

Step 1. (Disconnect external power) — If you performed the Power-Up Test in the last section, discoonect the power supply from the C3 now.

Step 2. (Install USB Drivers) — Before connecting your C3 to your PC for the first time, you need to install the FTDI Inc. USB VCP drivers. They are included with the Propeller Tool programming software, free from the Downloads link at www.parallax.com/propeller. The FTDI Windows driver installer is also available on its own from www.parallax.com/usbdrivers; for other options check www.ftdichip.com/FTDrivers.

If the FTDI drivers are installed, the moment you insert the USB cable into the C3 for the first time even with the power off, the PC will recognize and install the proper FTDI Inc. VCP on your PC. This is normal, and not to worry. When the installation is complete you should see a small message in your Windows system tray to the right indicating that the hardware was installed properly and is ready for use. Curious users might even want to open up the PC's Control Panel → Hardware Settings and take a look at what virtual COM port number the USB is using.

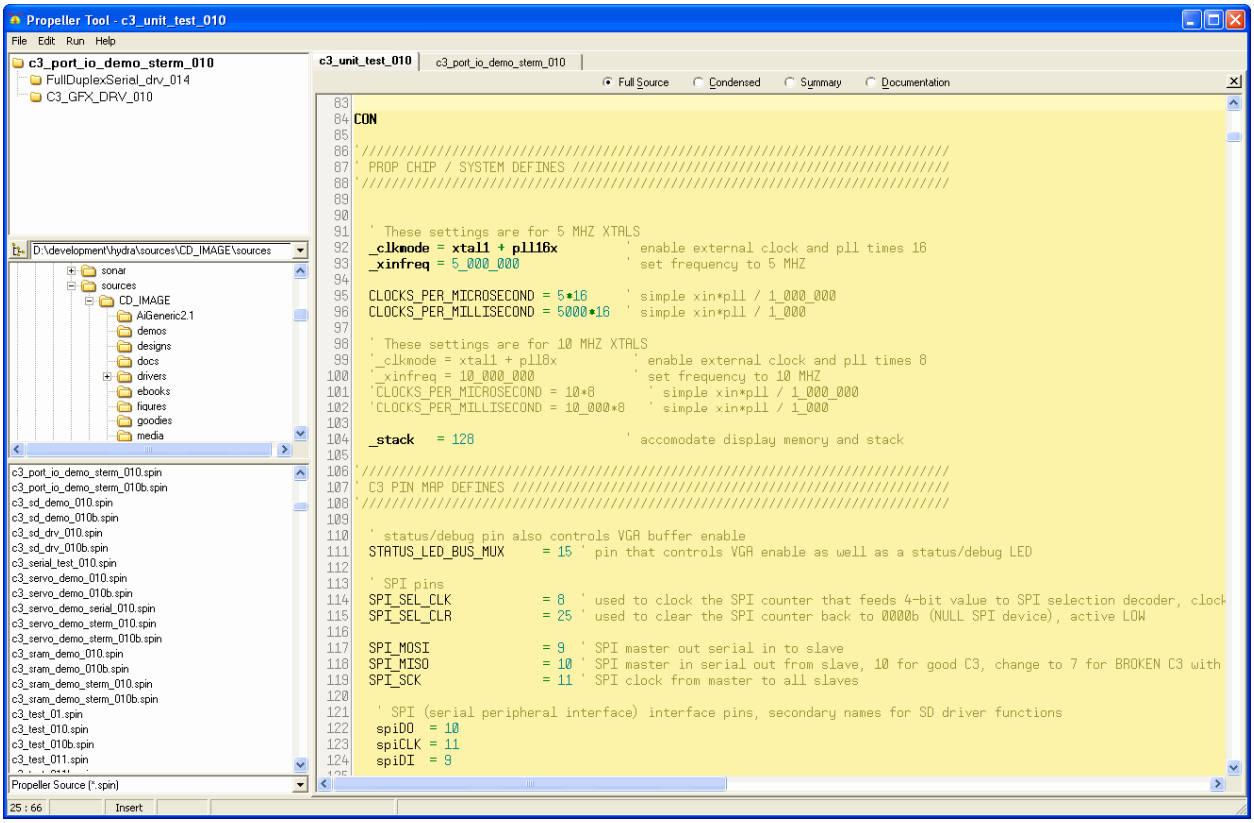
Step 3. (Prepping connections) — Hook up all the other connections as shown in Figure 1.2, EXCEPT the power supply.

Step 4. (Downloading the IDE) — if you haven't installed the Propeller Tool then you should do so now. You can download the latest copy of the tool the Downloads link on this page:

http://www.parallax.com/propeller

Download the latest Propeller Tool software and install the program. The program has a very detailed Help system and reference guide, please review that.

Figure 1.7 — The Propeller Tool up and running.



Step 5. (Running the IDE) — Once you are up to speed on the IDE, it is similar to any compiler tool you have used before. Launch the tool and you should see something similar to the screen shot shown in Figure 1.7. Next, we need to load the test program into the IDE.

Step 6. (Downloading the C3 software and demos to your hard drive) — Log onto the Parallax FTP site containing the source tree for the Propeller C3. The FTP site is anonymous, so all you have to do is click on the link below in your browser:

ftp://ftp.propeller-chip.com/PropC3

The directory structure is shown below:

PropC3	- Root directory.
Docs	- Contains documents relating to the C3, datasheets, etc.
Sources	- Contains source code and examples from this manual.
Games	- Contains games that have been ported or originals.
Apps	- Contains applications, languages, and other apps for the C3.
Designs	- Contains designs for the C3 including schematics and gerbers.
Tools	- Contains tools and programs for the C3.
Media	- Contains any extra media for the C3 or videos, audio, etc.
Goodies	- Contains any goodies that are special.
UPDATE_LOG.TXT	- Changes to the FTP directory are logged here.

Next, drag and copy the Sources\ directory to your hard drive (this will take a few moments to copy). If you have a very fast internet connection and don't mind waiting a little longer you can drag the entire root directory PropC3\ to your hard drive at once, so you don't have to copy any files later on required. But, for now the Sources\ sub-directory will do. Also, I suggest placing it in a directory called PropC3\ even if you don't drag the entire root directory to your hard drive to keep your directory structure consistent with what is expected in this manual.

Step 7. (Loading the test suite into the Propeller Tool) — Now that we have the source tree on your hard drive, let's load the test suite into the Propeller Tool. From the Propeller Tool's main menu, <Select File → Open>, navigate into the Sources\ directory on your hard drive and load the following file into the IDE:

c3_unit_test_010.spin

This is called a **"top object file"** in Spin-speak. When you compile the program it will actually refer to other files within the same directory and load those as well, but you won't have to worry about that—the IDE will handle it for you, identify the required files and load them from the source directory.

Step 8. (Compiling and downloading the program to the C3) — At this point, we are ready to compile and download the program to the C3. Make sure the C3 is plugged into the PC via the USB cable, and the C3 is powered ON (switch to the left). To compile and download the program to the C3, simply press the <F11> key (you can also, select <Run → Compile Current → Load EEPROM> from the main menu). You should see a dialog box pop up and display the status as the program compiles and downloads to the C3. After the download, the C3 will automatically re-boot and the test suite will run as it did out of the box.

This concludes the Quick-Start guide, now let's move onto the Propeller C3 design itself.

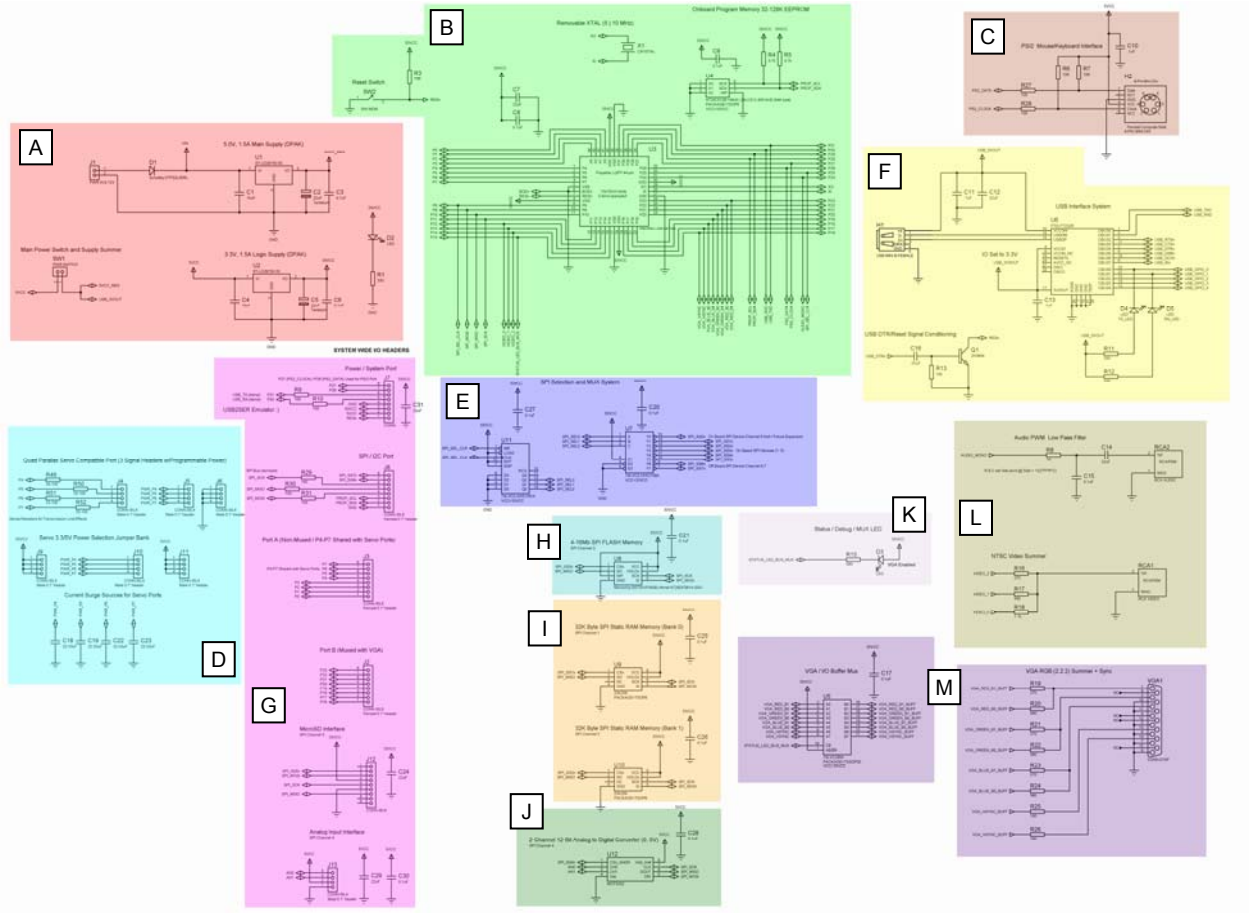
2 Hardware Overview

In the section of the manual we are going to review each of the hardware sub-systems of the C3 and see how they work. Even if you're not an electrical engineer or familiar with electronics, you should be able to follow most of the material and get something out of it. It can't hurt to understand the hardware to help write better software!

The approach I am going to take is we are going to start with a high level overview of the C3 for a few paragraphs and then drill down to each sub-system, look at the schematic and I will bring your attention to anything interesting. Additionally, realize that the C3 like any Propeller development board follows a lot of the same design rules, so if you have a Propeller Demo Board, HYDRA, or other Propeller-based device, many design elements should be familiar to you. With that said, let's begin…

The Propeller C3's design was predicated upon creating a very small, credit card sized board to show off what the Propeller can control in a system-wide design. Thus, the C3 is truly complete with extra RAM, FLASH, IO, and SD drive all built in—it's a true credit card sized computer with an arsenal of peripherals. On the other hand, we wanted all drivers written for the Propeller chip and/or other boards to be easy to port, thus we had to design a bus system that had minimal IO impact, but allowed any number of SPI based devices to be access over a common bus. Thus, in the tradition of old 8-bit computers and the IBM PCs, the Propeller C3 has a simple SPI bus that is used to interact with all its new peripherals such as RAM, FLASH, SD card, A/D, and more. This flexibility not only allows the porting of other applications that don't use these features to be very easy, but opens up a whole world of applications.

Figure 2.1 — The system-wide Propeller C3 schematic-



Referring to Figure 2.1, this is the complete schematic of the C3 in a single view. Don't worry if you can't see it clearly, I just want you to get an idea of what's what, so I have color-coded the various areas of the design, so I can point them out in the paragraphs that follow. A high resolution bitmap of the image can be found in the FTP files in the following location, so you can see more detail (color coded and b/w version):

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_lrg_colored_01.png
PropC3 \ Designs \ Schematics \ prop_c3_rev_a_lrg_01.png

Referring to the figure, starting from top left and moving left to right, top to bottom, we have the following sub-systems:

A. Power Management (top left; red color coded) — A pair of ST LD29150 voltage regulators one for the 5.0 V and the other for the 3.3 V supply. These regulators can source up to 1.5 A of current.

!	**The Propeller C3 Is designed to accept power from the USB port OR from an external power supply. Do not simultaneously supply power from more than one source. Before supplying external power be sure to remove the USB cable, otherwise damage to the C3 and your computer may occur.**

B. Propeller P8X32A (top middle, green color coded) — The Propeller chip along with bypass capacitors, 64 KB EEPROM, reset switch, and crystal input.

C. PS/2 Port (top right, brownish color coded) — This is the standard PS/2 port accepting keyboards and mice. There is only one PS/2 port on the C3. If you don't use the PS/2 port then the two IO lines can be used for other things, so they are exported to the system headers on the right side of the board.

D. Servo Port Header (middle left, cyan color coded) — The Propeller C3 can only source and sink so much current, so it can't really drive servos without external power. However, if you are careful and heavily bypass the power lines, you can directly drive servos as long as the power supply spikes don't reset the Propeller. The C3 has 4 servo port compatible headers with the standard 3-signal pin out (signal, power, ground). The control signals for each port are connected to P4 thru P7 on the Propeller. Also, each servo port's power is jumper selectable between 3.3 V and 5 V and heavily bypassed for current spikes. Finally, each servo control line has an inline series current limiter resistor which not only protects the PropellerIO from any inductive kicks, but acts as a transmission line dampener, so you can run larger 1–2 foot cables from the header and the signal integrity will remain reasonable. Of course, no one said you have to control servos with these headers, many Parallax customers simple like the clean 3-pin servo cables to use on their projects, thus we added the feature to the C3.

Adding a Bulk Capacitor for Driving Servos—Next to the 2.1mm ID, 5.5 mm OD, center positive power input header, there are two plated through holes designed for a large bulk capacitor. If you are going to drive servos then I suggest placing a 2200–4700 µF electrolytic capacitor across the + - ports and solder it under the board (ground to left, positive to right).

E. SPI Bus System (middle; violet color coded) — The majority of extra peripherals on the C3 are connected through a high-speed SPI bus that all the peripherals share. The SPI bus is capable of 25 MHz signaling, so it's very fast. Of course, parallel access is always faster, but accessing a parallel memory for example can use up 20 IO lines just for a 16 KB SRAM! The SPI protocol requires 4 signals per device: serial out, serial in, clock, and a chip select. The first three can be shared, but the last—chip select—can't. This signal needs to be generated independently for each device. A trick is used on the C3 to reduce the IO impact of these chip select requirements. Instead of using a separate IO line for each chip select, we could use 2 of them and a decoder to select 1 of 4, but we need more chip select lines, thus using 3 IO lines would give us 1 of 8, but that single line was just too much. So, the trick is that we use 2 lines to select up to 16 devices (actually 8 are only needed). The 2 IO lines control a 4-bit counter, and its reset. Thus, as we clock the counter it counts then the output of the counter routes into a 3-8 decoder. Of course, only a single SPI device can be selected at once since they share a common bus, but this isn't a problem in most cases and this is how all bused computers work, one device uses the bus, and they share.

F. FTDI USB Serial UART (right side under PS/2 port; yellow color coded) — This is a standard FTDI FT232BL USB serial UART chip along with a little DTR reset circuit, so the Propeller can be reset via the serial interface before programming. The USB +5 V can also used to power the C3 when external power is not being applied. Do not draw more that 500 mA when running off of the USB port.

G. System-wide IO Headers (left side to right of servo headers; magenta color coded) — These are simply all the IO headers for the C3. They include the 2x16 rows of IO on the right side of the C3 board as well as the microSD header/slot and A/D port.

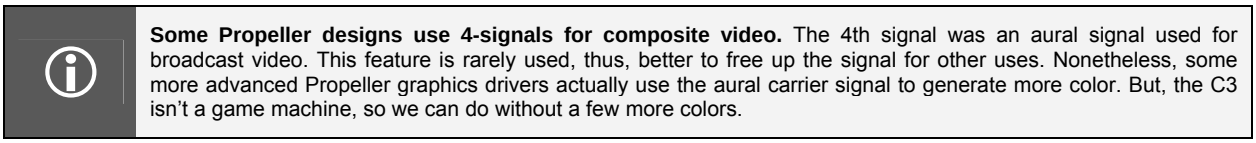
H. 1 MB SPI FLASH Memory (middle under the SPI system; color coded aqua) — The SPI FLASH on the C3 is a 8 Mbit / 1 MByte page organized FLASH memory from Atmel Inc. part #AT26DF081A-SSU. FLASH memories are somewhat interchangeable since they follow JEDEC standards, but each manufacturer likes to add their own little features which break the standard. Thus, on the C3 we have the Atmel part specifically, so I could write a driver that I know works.

I. 32 KB SPI SRAM Banks (middle under the FLASH memory; color coded orange) — The C3 has a pair of 32 KB SPI SRAMs giving a total of 64K bytes of external memory to work with. SPI SRAMs are relatively new technology and only a few vendors manufacture them in such large sizes. The SPI SRAMs used on the C3 are Microchip's 23K256.

J. 2-Channel 12-bit SPI Analog to Digital Converter (middle bottom under SRAMs; color coded dark green) — The C3 has a dedicated A/D converter. Although, it's possible to perform software A/D with the Propeller, the processing load and allocation of a cog to the task isn't a good use of the resource. Thus, the C3 has a Microchip SPI based MCP3202 2-channel, 12-bit A/D. It can sample two single ended signals (0..5 V) or one differential signal. The MCP3202 was chosen for its ease of use and availability; there are already drivers for it in the Parallax Object Exchange.

K. Status / VGA Enable LED (middle to right of FLASH memory; color coded light violet) — What's a development board without at least one LED! Well, the IO is so optimized on the C3, I couldn't even spare a single IO to dedicate to an LED. Therefore, I multiplexed the status LED with the VGA buffer enable which as its name states "enables" the VGA buffer to the VGA header. Thus, if you need to blink an LED, you can toggle this IO line, but be aware if you are using VGA at the same time, it will disable the display momentarily.

L. Composite Video and Audio (right middle; light brown color coded) — Standard Propeller composite video and audio circuits here. The composite video simply has a 3 resistor summing circuit and the audio has a PWM low-pass filter/integrator.



Some Propeller designs use 4-signals for composite video. The 4th signal was an aural signal used for broadcast video. This feature is rarely used, thus, better to free up the signal for other uses. Nonetheless, some more advanced Propeller graphics drivers actually use the aural carrier signal to generate more color. But, the C3 isn't a game machine, so we can do without a few more colors.

M. VGA System and Buffer (right bottom left under composite video; color coded purple) — The VGA output of the C3 is similar to other Propeller designs in that it uses 8 signals in parallel to control RGB along with Hsync and Vsync. However, the C3 (similarly to the HYDRA) has a tri-stateable buffer (a standard 74LVC245A, to the left of VGA header on schematic) from the Propeller to the VGA allowing the VGA connector (if attached) to be electrically removed from the Propeller IO bus (IO P16..P23). This allows non-VGA applications to re-use and multiplex Propeller IO signals P16..P23.

This concludes the brief review of each sub-system. We will go much further into detail in the sections below, but hopefully now you have an idea of how everything fits together.

2.1 System-wide Pin-out Reference

In this section, we are going to briefly review the Propeller C3 IO pin outs in tabular format for ease of access. Also, this will help as you review each of the system schematics in the sections below.

Figure 2.2 — Propeller C3 PCB layout

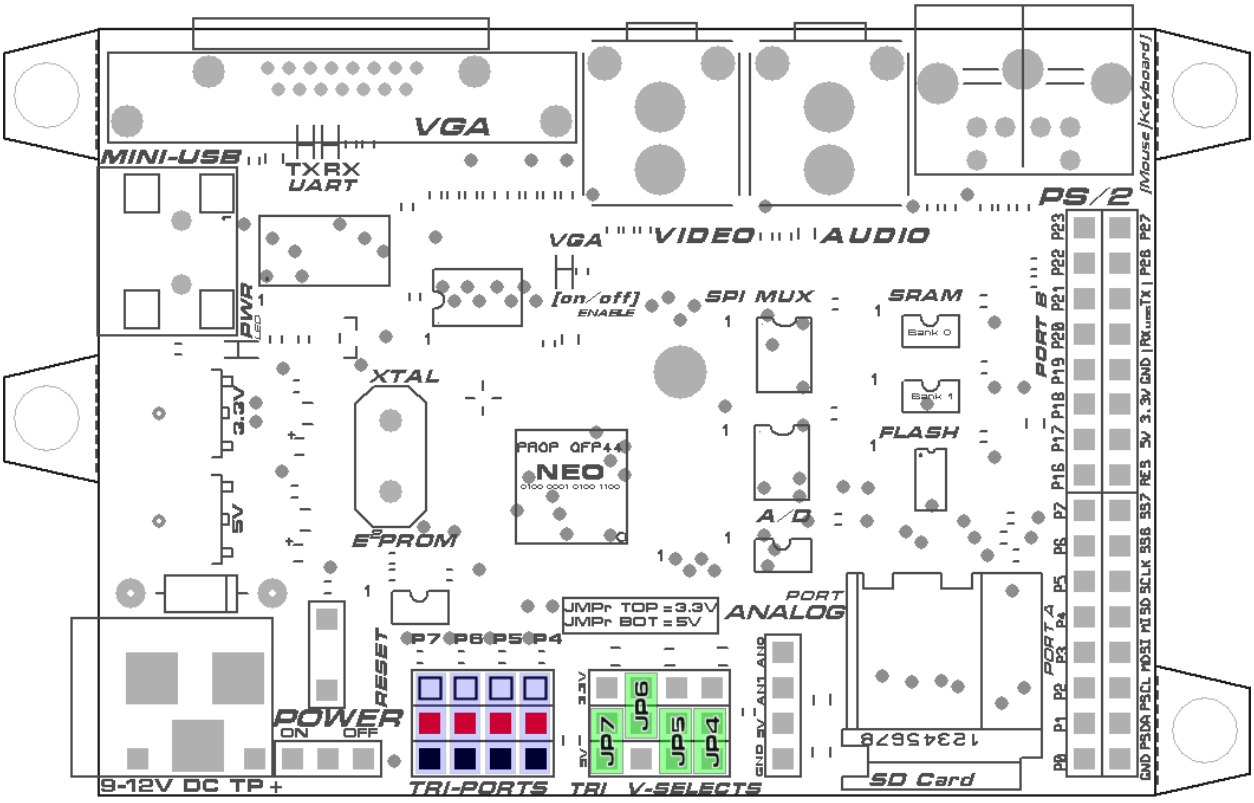


Figure 2.2 depicts a mechanical layout of the Propeller C3 PCB. Let's briefly review the headers and jumpers and review the signals for each header from the Propeller chip.

2.1.1 Tri-Ports Headers

The C3 supports (4) servo port female connectors (Parallax part #805-00001). These are convenient cables that are commonly used to route control and power signals to servos, but can be used for anything you wish. They are color coded with **[white, red, black]** mapping to **[signal, power, ground]**. This same signal order is used on the C3 as depicted by the same colors, so you can plug them as shown in Figure 2.2 above the **"Tri-Ports"** label with the blue tint. There are 4 vertically oriented servo connector ports with ground always on the bottom. Next, the middle pin moving upward is power (3.3 V or 5.0 V), and finally the topmost pin on each header is the signal pin. These signals come from the Propeller chip, **P7..P4** respectively left to right on the Tri-Ports. So, all you have to do is plug a cable into one or more of the 4 headers in a vertically oriented way, with ground at the bottom and then you can signal on Propeller IO pins **P7..P4** to control your device or servo.

Additionally, you can select the voltage on each of the servo ports and set it as 3.3 V or 5.0 V. This is accomplished by setting the jumpers to the right of the Tri-Ports themselves shown in green.

2.1.2 Tri V-Selects Jumpers

Each of the servo ports left to right, **P7..P4** has a power pin used to drive the device. The power pin can be set for 3.3 V or 5.0 V. This is accomplished via the jumpers to the right of the Tri-Ports labeled **"Tri V-Selects"**. These jumpers are individually labeled **JP7..JP4** as shown in Figure 2.2 tinted in green. When a particular jumper is in the topmost position it sets the associated power on the servo header to 3.3 V and when in the bottom position it sets it to 5.0 V. For example, in Figure 2.2, the jumpers are set as follows:

JP7 — Bottom position, Servo Port P7 (left most on the Tri-Ports) will have **5.0 V** at its power pin.
JP6 — Top position, Servo Port P6 (2nd from left on the Tri-Ports) will have **3.3 V** at its power pin.
JP5 — Bottom position, Servo Port P5 (2nd from right on the Tri-Ports) will have **5.0 V** at its power pin.
JP4 — Bottom position, Servo Port P4 (rightmost on the Tri-Ports) will have **5.0 V** at its power pin.

Of course, most servos are 5.0 V, so you would put all the jumpers in the lower position in most cases.

2.1.3 Analog Port Header

To the right of the Tri V-Selects and left of the SD card header is the Analog Port Header. This is where you can sample analog signals 0..5 V and they are fed into the MCP3202 SPI 2-channel A/D. The signals on the header are oriented top to bottom as shown in Table 2.0.

Table 2.0 – Signals top to bottom on Analog Port Header.

Signal Name	Propeller IO(Pin)	Description
AN0	N/A	Connected to analog input channel 0 of MCP3202
AN1	N/A	Connected to analog input channel 1 of MCP3202
5.0V	N/A	Connected to system power 5.0 V
GND	N/A	Connected to system ground (analog reference)

You must always reference the analog inputs from the ground pin GND. Additionally, you can power your external analog device from the +5.0 V pin if you wish. Either way, the A/D converter expects a voltage range of [0..5 V] at its inputs which will be converted into an integer [0..4095]. If you need more range then use a voltage divider or scaler.

2.1.4 Port A Header

The Port A header is on the right side of the PCB, next to the SD card socket. It consists of 8 signals, connected to Propeller IO pins P7..P0 (top to bottom) as shown in Table 2.1. Also, note that P7..P4 is shared with the servo port signals, so if you drive something connected to the servo port headers, make sure to disconnect anything sharing P7..P4 on the Port A header.

Table 2.1 — Signals top to bottom for Port A Header (General IO Port).

Signal Name	Propeller IO | Pin	Description
P7	P7 (4)	General IO, also connected to servo port header P7
P6	P6 (3)	General IO, also connected to servo port header P6
P5	P5 (2)	General IO, also connected to servo port header P5
P4	P4 (1)	General IO, also connected to servo port header P4
P3	P3 (44)	General IO
P2	P2 (43)	General IO
P1	P1 (42)	General IO
P0	P0 (41)	General IO

2.1.5 Port B Header

The Port B header is right above Port A and consists of 8 signals connected to Propeller IO pins P23..P16 top to bottom as shown in Table 2.2. These IO pins are also shared by the VGA port on the C3, so if you are driving a VGA monitor then these IO signals will reflect those signals. Additionally, there is a buffer on the VGA signals in front of the VGA header itself. You can disable this buffer so that even if you have a VGA monitor connected to the VGA port, the impedance will not disturb the IO pins on Port B. To enable/disable the VGA buffer you use Propeller IO pin **P15** which is called **STATUS_LED_BUS_MUX** in the Propeller C3 schematic. Setting this signal **low enables** the buffer, **high disables** it (more on this when we discuss hardware in detail).

Table 2.2 — Signals top to bottom for Port B Header (General IO Port shared with VGA).

Signal Name	Propeller IO(Pin)	Description
P23	P23 (26)	General IO, also connected to VGA_RED_B1.
P22	P22 (25)	General IO, also connected to VGA_RED_B0.
P21	P21 (24)	General IO, also connected to VGA_GREEN_B1.
P20	P20 (23)	General IO, also connected to VGA_GREEN_B0.
P19	P19 (22)	General IO, also connected to VGA_BLUE_B1.
P18	P18 (21)	General IO, also connected to VGA_BLUE_B1.
P17	P17 (20)	General IO, also connected to VGA_HSYNC.
P16	P16 (19)	General IO, also connected to VGA_VSYNC.

Port A and B headers are not to be confused with I/O port designations on the Propeller chip internally. The design has a port "A" which refers to the normal 32 bits of I/O on the Propeller chip, and a Port "B" which refers to an additional 32 bits of I/O for future expansion. This has nothing to do with Port A and B headers on the C3, they are just names on the PCB.

2.1.6 SPI / I²C Header

The SPI / I²C header is to the right of the Port A header along the right edge of the PCB. The header consists of 8 signals that export out the SPI bus (plus 2 unused SPI chip selects), I²C bus, and ground, top to bottom as shown in Table 2.3 below. The SPI bus logic is not built into the Propeller, but it is created by the program running on the Propeller, which sends data out the SPI bus one bit at a time. The I²C bus, once again is not implemented in discrete logic, but the Propeller uses software to follow the I²C protocol to boot the EEPROM, thus we have exported out these two IO pins, so you can place other I²C devices on this bus. But, as usual, in both cases of SPI and I²C you must bit-bang the protocols yourself.

Table 2.3 — Signals top to bottom for SPI / I²C Header.

Signal Name	Propeller IO (Pin)	Description
SS7	NA (generated internally)	SPI select channel 7 (active low)
SS6	NA (generated internally)	SPI select channel 6 (active low)
SCLK	P11 (12)	SPI clock
MISO	P10 (11)	SPI master in slave out (into Propeller)
MOSI	P9 (10)	SPI master out slave in (from Propeller)
PSCL	P28 (35)	Serial clock out of Propeller
PSDA	P29 (36)	Serial data in/out of Propeller
GND	Ground	System ground

The SPI select signals labeled SS7 and SS6 are generated by the SPI bus hardware and are external to the Propeller. They actually connect in the schematic to the output of the 74LVC138A decoder pins Y7, Y6 at pins 7,9 respectively of the 138A.

2.1.7 System / Power Header

The System/Power header is to the right of the Port B header along the right edge of the PCB. The header consists of 8 signals that export out the PS/2 signal P27, P26, the USB serial TX, RX, power, ground and system reset as shown in Table 2.4 below.

Table 2.4 — Signals top to bottom for System / Power header.

Signal Name	Propeller IO (Pin)	Description
PS2_DATA	P26 (33)	PS/2 serial data non-inverted
PS2_CLOCK	P27(34)	PS/2 serial clock non-inverted
TX (USB)	P31 (38)	USB TX out, Propeller RX in
RX (USB)	P30 (37)	USB RX in, Propeller TX out
GND	Ground	System ground
3.3V	NA	Power supply 3.3 V
5.0V	NA	Power supply 5.0 V
RES	RESn (7)	System reset (Active low)

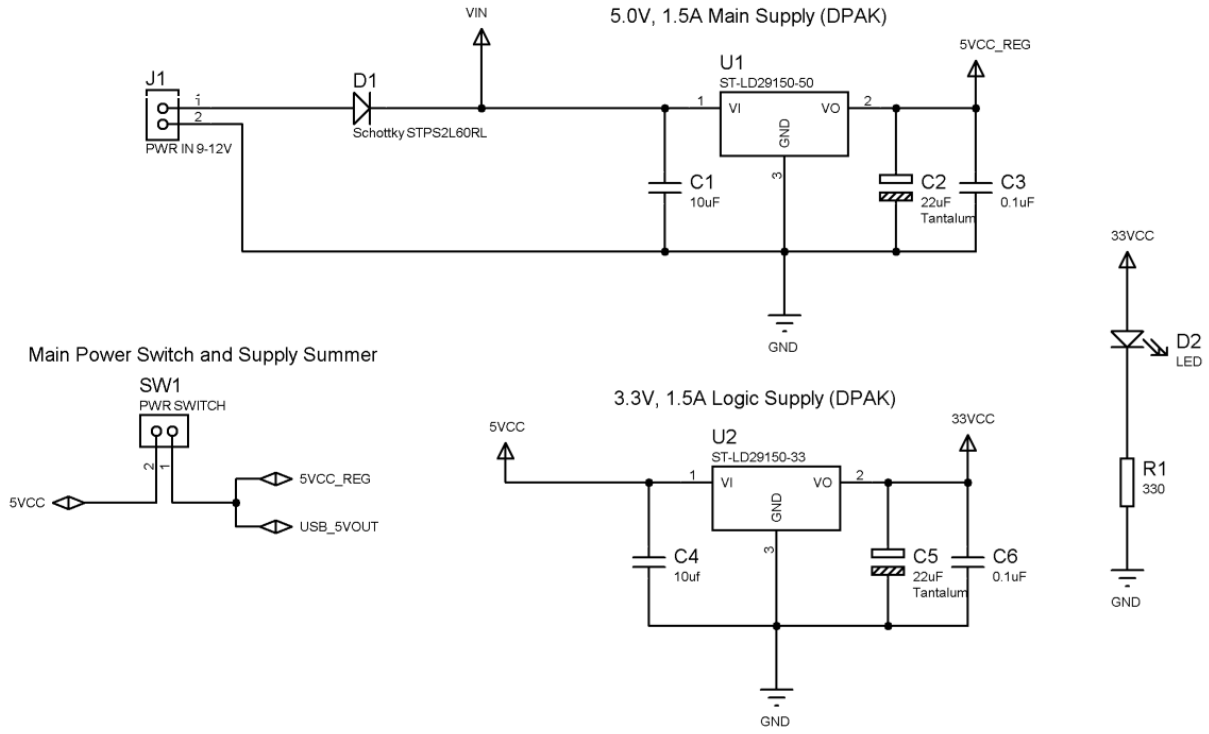
In the next sections, we will cover each sub-system in detail with a close up schematic and discussion of the circuit operation and any software considerations.

2.2 Power Management

The power for the Propeller C3 is sourced either by an external 9—12 VDC wall adapter with a 2.1 mm ID, 5.5 mm OD, center positive port or pulled from the USB port. The C3 can be powered from either but not both. Additionally, the C3 can draw a lot of current with all IO devices inserted and even more when driving servos or large current devices. Thus, if you do wish to power the C3 by a USB cable, then make sure that the USB port is high power and can supply 500 mA of current. If you try to draw too much current from a USB port then the port will disable or shut down and a hardware reset is usually required. Additionally, try not to connect highly inductive loads to the C3 when USB powered, and don't connect both external power and USB power at the same time. That said, you can easily power your C3 from a laptop as long as you aren't driving high-current devices from it. Now, let's take a look at the power supply schematic as shown in Figure 2.3 below. Also, you can find a high-resolution copy of schematic on the FTP site in:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_power.png

Figure 2.3 — The power management circuitry for the Propeller C3



The Propeller C3 Is designed to accept power from the USB port or from an external power supply. Do not simultaneously supply power from more than one source. Before supplying external power be sure to remove the USB cable, otherwise damage to the C3 and your computer may occur.

Referring to Figure 2.3, power is either fee from the USB 5 V supply or the external port at **J1** through a Schottky protection diode **D1** into the 5 V regulator **U1**. The output of U1 and the 5 V supply from the USB connector are both directly connected to the input of the 3.3 V regulator **U2**. Both regulators are high current LDOs (Low Drop Out) from ST-Micro LD29150 series. You can find a data sheet for them on the FTP site here:

PropC3 \ Docs \ Datasheets \ ST29150.pdf

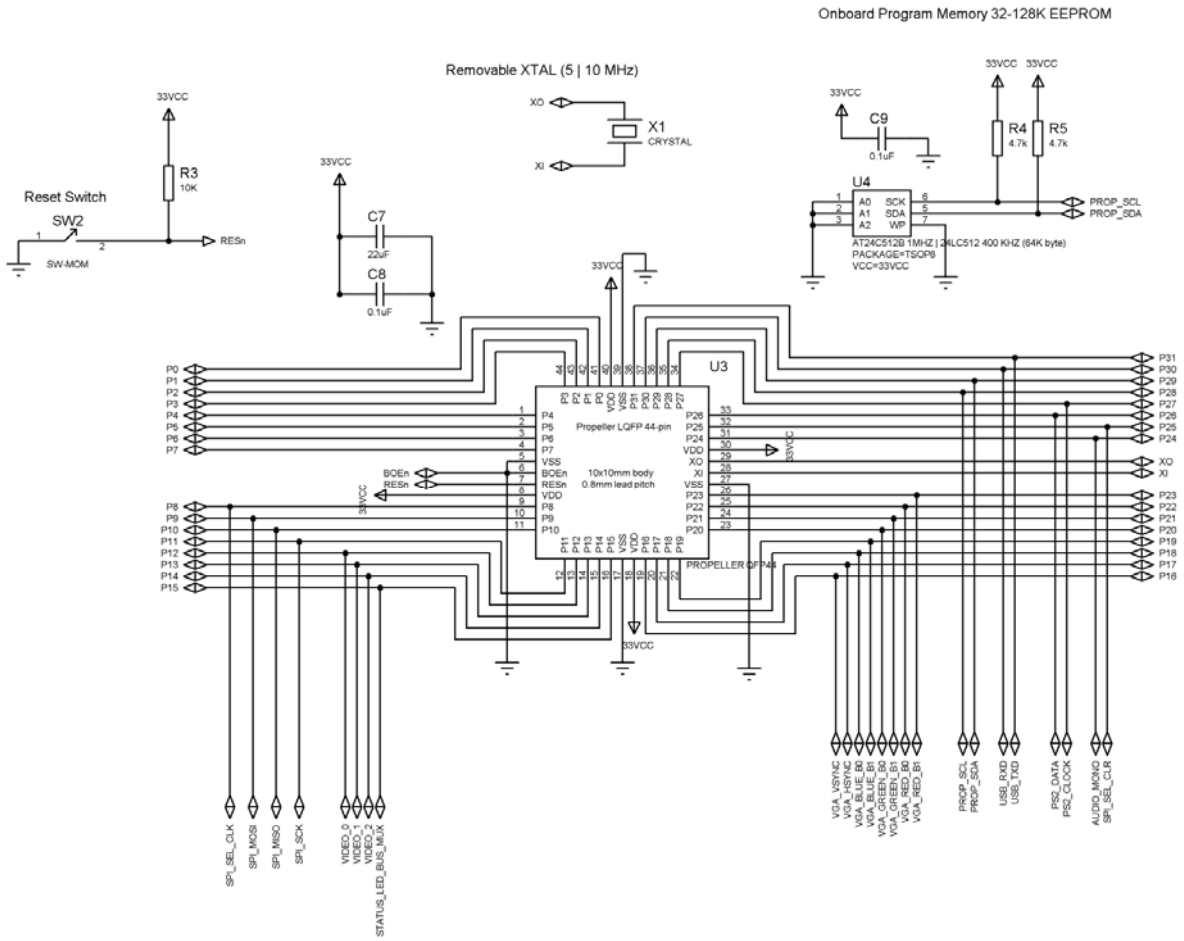
Finally, power on is indicated by LED **D2** with current limiter resistor **R1**.

2.3 Propeller Processor System

The Propeller chip on the C3 is a standard QFP44 pin package oriented with pin 1 to the bottom right of PCB. The processor system on the C3 includes the standard requirements such as an EEPROM for program storage (64 KB in this case), reset circuit, crystal, and bypass capacitors for power smoothing. Some special design elements on the C3 are extra large SMT pads for the C3 to increase heat dissipation and keep the Propeller cool for over clocking. Additionally, the Propeller has a rather large 22 µF bypass capacitor along with a 0.1 µF capacitor in parallel to reduce noise as well as well as supply large currents on demand, so the chip doesn't brown out. Let's take a look at the schematic shown below in Figure 2.4. Also, there is a high resolution copy of the schematic on the FTP site in:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_prop.png

Figure 2.4 — The Propeller C3's processing hardware



Referring to Figure 2.4, the Propeller is labeled U3, the reset logic at switch SW2 allows simple push-button reset. The crystal is socketed at X1, so you can remove and replace it with a 10 MHz, or the new popular 6.25 MHz for over clocking experiments. At U4 we see the boot EEPROM which is a Microchip 24FC512-I/ST 64 KB byte I^2C EEPROM. Of course, the Propeller only requires a 32 KB boot EEPROM, but having more space allows other assets to be stored in the EEPROM. You can find the data sheet for the EEPROM on the FTP site here:

PropC3 \ Docs \ Datasheets \ 24FC512.pdf

Now, let's take a look at all the signals coming from the Propeller chip. All of the standard IO signals are exported out of the chip on the schematic with their Propeller IO names, then a number of specialized aliases are made on the schematic to make the design easier to keep track of what's what.

On the left of Figure 2.4, you can see the SPI signals, as well as the video signals. Notice that instead of the usual 4 video signals, we have omitted the aural signal (since it's rarely used) and used this signal to control the status LED as well as the VGA buffer (STATUS_LED_BUS_MUX). Moving to the right of the figure, we see the VGA signals, as well as the I^2C, USB TX/RX, PS2, and finally a single audio signal, and the SPI_SEL_CLR signal. Table 2.5 below lists all the signals by class and gives a detailed description of each.

Table 2.5 — Propeller chip primary signals.

Signal Name	Propeller IO(Pin)	Description | Notes
SPI		**Propeller is master.**
SPI_SEL_CLK	P25 (32)	Used as the clock input into the SPI chip select logic to cause the 4-bit counter to count up.
SPI_MOSI(1)	P9 (10)	SPI master out slave in.
SPI_MISO(1)	P10 (11)	SPI master in slave out.
SPI_SCK(1)	P11 (12)	SPI clock.
SPI_SEL_CLR	P8 (9)	Used to clear the 4-bit counter in the SPI chip select logic.
Composite Video		**Standard Propeller video DAC signals.**
VIDEO_0	P12 (13)	Bit 0 of video signal.
VIDEO_1	P13 (14)	Bit 1 of video signal.
VIDEO_2	P14 (15)	Bit 2 of video signal.
VGA		**Standard Propeller VGA DAC signals. VGA also shared with Port B header.**
VGA_VSYNC	P16 (19)	TTL level vertical sync pulse.
VGA_HSYNC	P17 (20)	TTL level horizontal sync pulse.
VGA_BLUE_B0	P18 (21)	Analog Blue signal bit 0.
VGA_BLUE_B1	P19 (22)	Analog Blue signal bit 1.
VGA_GREEN_B0	P20 (23)	Analog Green signal bit 0.
VGA_GREEN_B1	P21 (24)	Analog Green signal bit 0.
VGA_RED_B0	P22 (25)	Analog Red signal bit 0.
VGA_RED_B1	P23 (26)	Analog Red signal bit 0.
LED Status / VGA Mux		**Used for both a status LED toggle and to enable/disable the VGA buffer.**
STATUS_LED_BUS_MUX	P15 (16)	Active low signal enables the VGA buffer and connects the VGA signals P23..P16 to the VGA header. Also, low lights the status LED.

I^2C		**Propeller is I^2C master normally during boot, but if pins are placed into input mode then an external device can actually communicate to the on-board I^2C as master.**
PROP_SCL(2)	P28 (35)	I^2C clock w / 4.7 kΩ pull-up to 3.3 V.
PROP_SDA(2)	P29 (36)	I^2C data w / 4.7 kΩ pull-up to 3.3 V.
Serial/USB Serial		
USB_RXD	P30 (37)	Receiver to USB UART from TX of Propeller.
USB_TXD	P31 (38)	Transmitter from USB UART into RX of Propeller.
Audio		**Assumed that this will be driven with a PWM signal of some kind.**
AUDIO_MONO	P24 (31)	Mono audio PWM signal with integrator and filter before RCA output.

Note 1 – The Propeller chip is the master in the Propeller C3 design and initiates all SPI traffic.

Note 2 – Standard electrical protocol for I^2C specifies that I^2C clock and data should be open collector, so either master or slave can pull them to ground. Thus, the 4.7 kΩ pull-up resistor is used in case the Propeller stops acting as a slave and there are other I^2C masters on the external header. They will require the pull ups for proper operation.

2.4 SPI Bus System

The Propeller C3 is a highly optimized design in both design and size respects. One of the issues with any microcontroller design is how to make best use of the IO resources while at the same time not drive yourself into a brick wall or creating a messy design. After the initial design of the core Propeller C3 features such as composite video, VGA, serial, I^2C, PS2, etc. there was hardly any IO left to access all the new devices we wanted to add to the Propeller C3 such as FLASH, SD, SRAM, A/D, and external devices.

The first decision was to go with an SPI design (serial peripheral interface) rather than I^2C. The reasoning is that although SPI requires a separate clock, it's MUCH faster and worth the extra signal line. Therefore, we needed at least 3 lines for the base SPI bus interface **MOSI**, **MISO**, and **SCLK**. So far, so good, but I only had 2 signals left on the board after this allocation! And SPI devices need a chip select line. I could use the 2 remaining signals to feed a decoder, but that would give me only 4 chip select lines, I needed at least 8 to be able to access all the onboard devices as well as give the user at least 2 exported chip select lines (**SPI_SS7**, **SPI_SS6**) for connecting more devices.

The choice was pull a signal from something else, or multiplex the SPI bus with one of the signals on P7..P0—which I just couldn't bring myself to do. That nice 8-bit port had to be left alone. So, I thought long and hard how to select any number of devices with only 2 lines and I came up with a very clean concept; the use of a counter and a decoder.

One of the most interesting stories about IO usage occurred during the design of the Atari 2600 or VCS. Back in the 1970's the 6502 microprocessor reigned supreme (in fact there are millions still sold today). At $25 a unit the 6502 was the basis for the Apple, Commodore 64, Atari computers, as well as the Atari 2600. The 2600 was based on a variant of the 6502, the 6507 which had only 13 address pins instead of 16, thus it could only address a total of 2^{13} = 8K. The designers decided to export only 12 of these address lines to the cartridge port, and split the 8 KB address space into 4 KB for internal RAM, and 4 KB for external ROM games. This was one of the largest mistakes in history. Very quickly, programmers used up the 4 KB cartridge memory and needed the extra address line. Atari designers saved about 0.5 pennies not exporting that address line, but lost millions if not tens of millions in more complex "bank switched" cartridges to overcome the oversight..

One signal would be used to clock a 4-bit counter (U11; 74LVC161A), the output of the counter would be fed into a 3-8 bit decoder (U7; 74LVC138A) then as the count increased the decoder would enable one

of n outputs (active low). These outputs would then be directly connected to the active low chip select line of each and every SPI device on the bus. Finally, to reset the counter, the other remaining IO line feeds the active low reset of the counter (SPI_SEL_CLR) and this allows you to very quickly reset the SPI select to device channel 0. If you are interested in learning more about the counter and decoder, their datasheets can be found on the FTP site here:

PropC3 \ Docs \ Datasheets \ 74LV161.pdf
PropC3 \ Docs \ Datasheets \ 74LV138.pdf

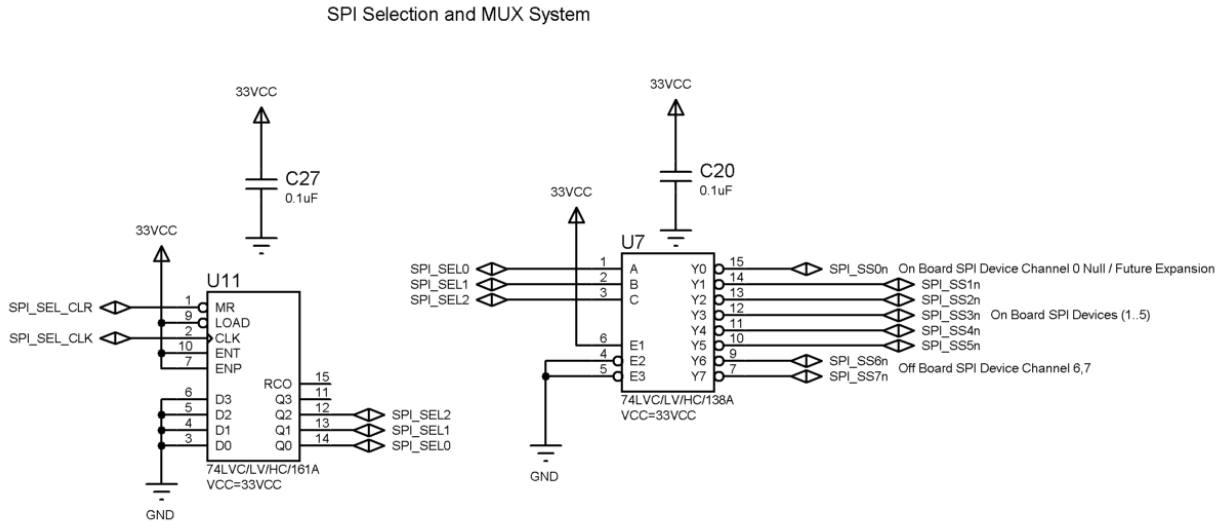
So, you might be thinking, "Isn't that slow to have to count up to the SPI channel each time?" The answer is no. You can clock the counter at 100 MHz if you wish, so it's nearly instantaneous. Moreover, once a device is selected you tend to use it for a long time, thus the selection process doesn't have effect on speed.

With this design, we could actually address 16 devices on the SPI bus if we used a 4-16 decoder, but I opted to use a 3-8 which allows all onboard devices to be accessed along with 2 more external devices on the SPI/I^{2}C header.

Of course, only a single SPI device can be accessed at once, since they share the same bus, but unless we have n parallel buses there is no way around this. Let's take a look at the schematic shown below in Figure 2.5. Also, there is a high resolution copy of the schematic on the FTP site in:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_spibus.png

Figure 2.5 — The SPU bus and multiplex logic.



The circuit is rather simple, the Propeller feeds the counter U11 (74LVC161A) with the clock and clear signals at pins 1,2 respectively. Then the counter counts and its 4-bit output is on pins Q3..Q0 at pins 11...14. Only the lower 3 bits are used: Q2..Q0 labeled SPI_SEL2..SPI_SEL0 which are fed into the decoder U7 (74LVC138A) at inputs A,B,C on pins 1,2,3 respectively. Thus, the counter as it counts from 0...7 causes the SPI select lines on the decoder Y0..Y7 to assert (active low). These signals are referred to in the schematic as SPI_SS0n..SPI_SS7n. The "n" is just to remind us they are active low, and are omitted in discussions from time to time. Therefore, to select any SPI device the algorithm is:

- Reset the counter by asserting SPI_SEL_CLR LOW then back high.
- Clock the counter n times by raising and lowering SPI_SEL_CLK.

- At this point, the appropriate SPI select line will be asserted from the decoder, you are free to access the SPI device on the bus.

Here's an actual Spin version of the function that selects an SPI channel on the Propeller C3:

```
PUB SPI_Select_Channel( channel )
{{
This function sets the active SPI channel chip select on the SPI mux, this is
accomplished by first resetting the SPI counter that feeds the SPI select decoder,
then up counting the requested number of channels.

PARMS:

channel : channel 0 to 7 to enable where the channels are defined as follows
          0 - NULL channel, disables all on/off board devices.
          1 - 32K SRAM Bank 0.
          2 - 32K SRAM Bank 1.
          3 - 1MB FLASH Memory.
          4 - MCP3202 2-Channel 12-bit A/D.
          5 - Micro SD Card.
          6 - Header Interface SPI6.
          7 - Header Interface SPI7.

RETURNS: nothing.

}}

  ' requesting channel 0? If so, easy reset
  if (channel == 0)
    ' clear the 161
    OUTA[SPI_SEL_CLR] := 0 ' CLR counter
    OUTA[SPI_SEL_CLR] := 1 ' allow counting
    return

  ' else non-null channel, count up to channel...

  ' first reset the SPI channel counter
  ' clear the 161
  OUTA[SPI_SEL_CLR] := 0 ' CLR counter
  OUTA[SPI_SEL_CLR] := 1 ' allow counting

  ' now increment to requested channel
  ' clock the 161
   OUTA[SPI_SEL_CLK] := 0

   repeat channel
     OUTA[SPI_SEL_CLK] := 1
     OUTA[SPI_SEL_CLK] := 0

' end SPI_Select_Channel
```

As you can see, certain channel numbers have been assigned to the various SPI devices on the Propeller C3; they are listed in the next section.

2.4.1 SPI Channel Allocations

- 0 — NULL channel, disables all on/off board devices.
- 1 — 32 KB SRAM Bank 0
- 2 — 32 KB SRAM Bank 1
- 3 — 1MB FLASH Memory
- 4 — MCP3202 2-Channel 12-bit A/D
- 5 — MicroSD Card
- 6 — Header Interface SPI6
- 7 — Header Interface SPI7

In the second section of this manual on software and programming the C3 we will see more of the SPI bus and related functions in their entirety; this is just to let you see how easy it is to use the SPI bus. Next, let's take a look at all the signals at a glance in Table 2.6 below.

Table 2.6 — SPI selection logic signals.

Signal Name	Propeller IO(Pin)	Description | Notes
SPI_SEL_CLK	P25 (32)	Used as the clock input of the 4-bit counter U11 to count up
SPI_MOSI	P9 (10)	SPI master out slave in
SPI_MISO	P10 (11)	SPI master in slave out
SPI_SCK	P11 (12)	SPI clock
SPI_SEL_CLR	P8 (9)	Used to clear the 4-bit counter U11
SPI_SEL2	NA (internal)	Bit 2 of counter feed to C input of 3-8 decoder U7
SPI_SEL1	NA (internal)	Bit 1 of counter feed to B input of 3-8 decoder U7
SPI_SEL0	NA (internal)	Bit 0 of counter feed to A input of 3-8 decoder U7
SPI_SS0n	NA (internal)	NULL channel, selects nothing
SPI_SS1n	NA (internal)	Selects SRAM bank 0
SPI_SS2n	NA (internal)	Selects SRAM bank 1
SPI_SS3n	NA (internal)	Selects 1MB FLASH memory
SPI_SS4n	NA (internal)	Selects A/D converter
SPI_SS5n	NA (internal)	Selects microSD card
SPI_SS6n	NA (internal)	Off board external device select channel 6
SPI_SS7n	NA (internal)	Off board external device select channel 7

2.5 USB Serial Communications

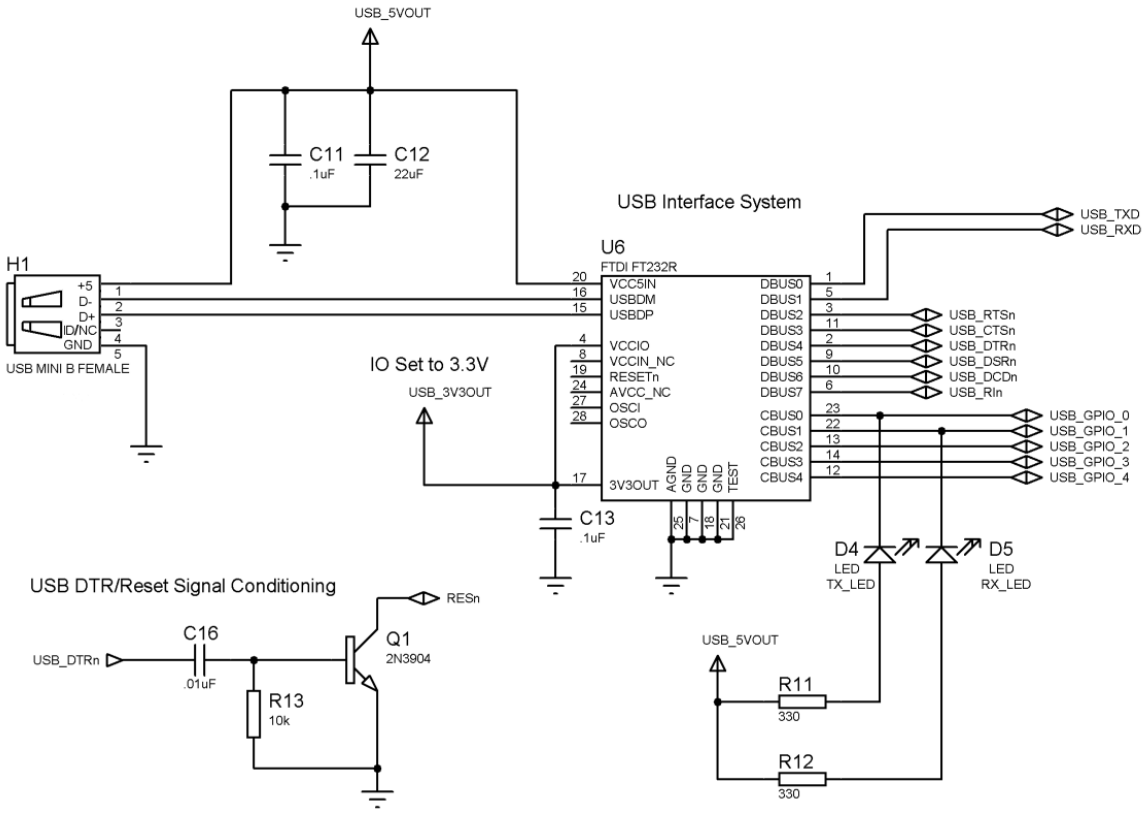
The serial port on the C3 uses a standard FTDI USB to serial UART **FT232R** at **U6**. This is the same chip used on the majority of Propeller based boards, and the stand-alone Parallax USB2SER and Prop Clip devices use it as well. There is nothing special about the FTDI chip other than the C3 can be powered by the 5 V input to the USB if there is no external power. The data sheet for the FTDI chip can be found on the FTP site here:

PropC3 \ Docs \ Datasheets \ FTDI232R.pdf

Other than that, the C3 uses the standard AC-coupled, DTR-based reset circuit. The schematic for the USB UART is shown below in Figure 2.6. There is a high resolution copy of the schematic on the FTP site in:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_usbserial.png

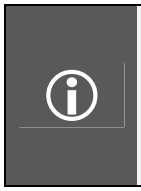
Figure 2.6 — The USB to serial UART circuit



The FTDI 232R at **U6** is connected to the Mini-B USB port with bypass/filtering capacitors on the +5.0 V supply. The FTDI chip is powered from the USB port itself and the IO is set to 3.3 V into the **VCCIO** pin of U6. The only other signals from the UART are the TX/RX LED indicators connected to **CBUS0**/**CBUS1** respectively. Finally, the UART TX line labeled **USB_TX** connects to the Propeller chip's RX line at **P31** and the UART RX line labeled **USB_RX** connects to the Propeller chip's TX line at **P30**.

The USB_TX, and USB_RX are exported out to the IO headers on the C3, so you can use the C3 as a pass-through ad-hoc USB to serial TTL converter.

Finally, there is the standard reset circuit based on the **DTR** output (Data Terminal Ready) of the FTDI chip labeled **USB_DTRn**. If there is a transient pulse on the DTR line it will pass through the capacitor at **C16** and forward bias the transistor **Q1**. This will in-turn cause **RESn** to be pulled to ground momentarily, resetting the Propeller chip. On the other hand, if the DTR signal is held high (latched up, etc.) then the reset will occur; however, as C16 charges, the right side will end up at 0 V or thereabouts, which will reverse-bias the transistor and the reset state will stop. Thus, this circuit is very safe and allows you to reset the Propeller via the DTR line, but won't lock up if the DTR is held accidentally.



If you want to create a PC-based serial application that controls DTR (or other signals) then you will need an appropriate driver. The Windows SDK has a serial driver that allows full control of the serial port's control signals. And under Linux the same kind of drivers are available for the TTY drivers. Also, some serial terminal programs allow you to control some of the serial handshaking lines as well. For example, the Parallax Serial Terminal program GUI allows you to toggle DTR as well as RTS (Request to Send) and view the state of the TX, RX, CTS (Clear to Send), and DSR (Data Set Ready) lines.

That's about all there is to the serial communications on the C3. Table 2.7 below lists the signals at a glance.

Table 2.7 — USB UART signals used in the Propeller C3 serial sub-system.

Signal Name	Propeller IO(Pin)	Description | Notes
USB_TX	P31 (38)	RX into the Propeller from PC
USB_RX	P30 (37)	TX from the Propeller to PC
CBUS0	NA	TX indicator from USB UART
CBUS1	NA	RX indicator from USB UART
USB_DTRn	NA	Controls system reset RESn on Propeller

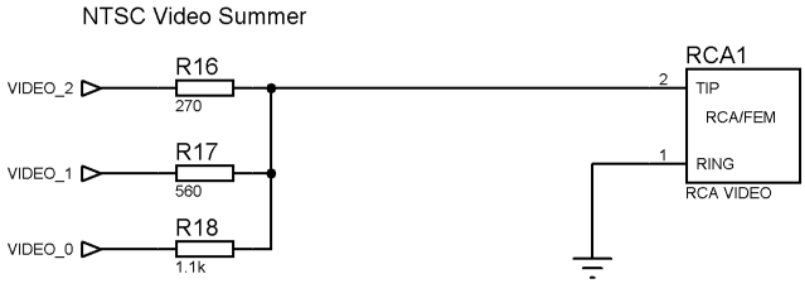
2.6 Composite Video

The composite video circuit for the C3 is the standard 3 resistor voltage summer circuit that is utilized by the numerous video drivers written for the Propeller chip. Programming a driver for the video hardware requires assembly language. The video hardware is beyond the scope of this manual, but if you are interested in learning more about writing video drivers for the Propeller chip then I suggest my book ***Game Programming for the Propeller Powered HYDRA*** (available from Parallax). It has the largest coverage of video programming on the Propeller chip.

With that in mind, let's take a look at the video schematic. The schematic for the video summer circuit is shown below in Figure 2.7 and there is a high-resolution copy of the schematic on the FTP site in:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_video.png

Figure 2.7 — The composite video circuit for the Propeller C3



The video circuit functions as a current summer where the voltage on each of the inputs **VIDEO_0..VIDEO_2** generate currents into the receiver's 75 Ω termination impedance; this results in a voltage swing of approximately 0 V (sync) to 1 V (bright white). The only thing you need to know about the video circuit is that if you are going to port another driver from another author or write one yourself, all you have to do is set the video pins correctly. The C3 uses **P12..P15** for video, that is the upper nibble of the second 8-bit port. Thus, when porting drivers make sure you set this correctly, and the associated mask. Also, remember the C3 doesn't use the aural sound signal on the last bit of the video nibble, thus you need to disable any aural signal that would be sent on P15.

Other than that, it takes a whole 30 seconds to make the changes to any video driver that might use a different 8-bit Propeller port and nibble. Table 2.8 below lists the signals and their associated Propeller IO pins.

Table 2.8 — Composite video summer circuit signals.

Signal Name	Propeller IO(Pin)	Description | Notes
VIDEO_0	P12 (13)	Bit 0 of video signal (LSB)
VIDEO_1	P13 (14)	Bit 1 of video signal
VIDEO_2	P14 (15)	Bit 2 of video signal (MSB)

2.7 VGA Video / IO Buffer

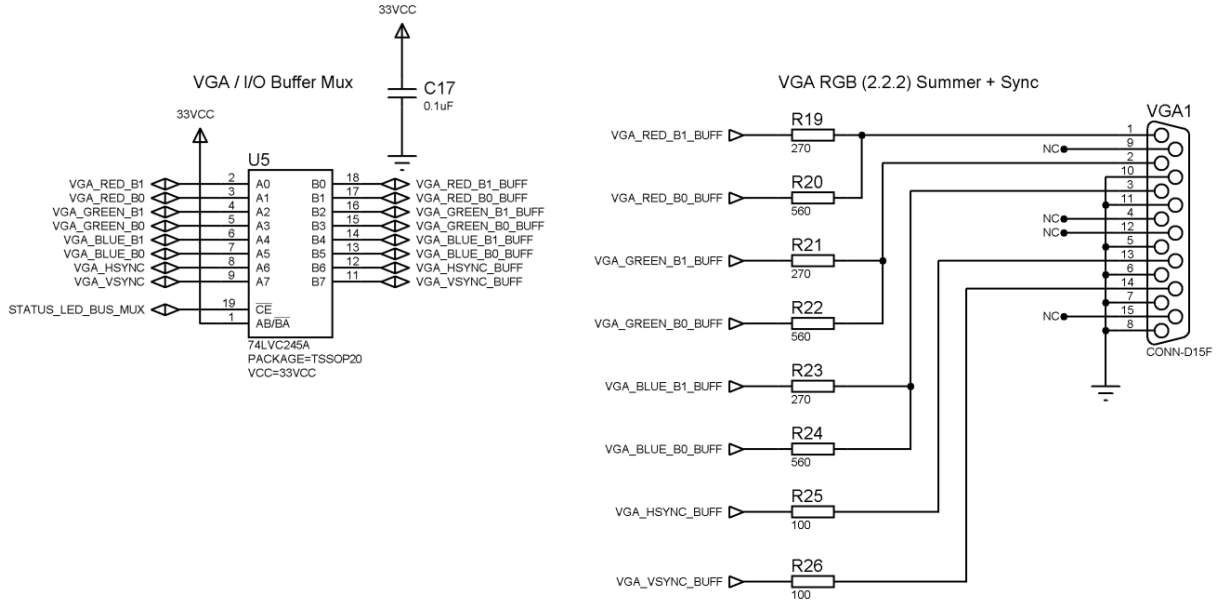
The VGA circuitry on the C3 is identical to the standard Propeller VGA design except that there is a **74LVC245A** buffer **U5** between the Propeller VGA signals and the VGA HD15 header. This is so we can share the VGA pins P23..P16 with other applications via the C3 header Port B. The buffer allows us to electrically remove the VGA HD15 connector (and the VGA cable connected to it potentially) from the IO pins themselves and use them without worry that there are other signals or impedances on them from a connected VGA monitor. Of course, this means you can only use either the VGA monitor or the IOs **P23..P16** on header Port B one at a time, but better than having to unplug a VGA monitor every time!

The buffer is can be tri-stated with the **STATUS_LED_BUS_MUX** signal connected to Propeller IO signal **P15** being asserted **low**. This also has the effect of lighting the "VGA Enable" LED next to the buffer chip itself on the PCB.

So, if you want to use the VGA output of the Propeller/C3 make sure to assert P15 to low. If you want to use the Port B header as standard IO, make sure to assert P15 to high. Aside from the buffer, the VGA display works like other Propeller designs, the video unit in any cog generates a stream of bytes, each byte represents RGB in 2:2:2 format and the sync signals. A series of 3 summing circuits sum the RGB signals respectively and the sync signals are injected into HSYNC and VSYNC directly on the VGA monitor. The schematic for the VGA video circuit is shown below in Figure 2.8 and there is a high resolution copy of the schematic on the FTP site in:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_vga.png

Figure 2.8 — The schematic of the buffer and VGA summing circuit



Referring to the circuit, the VGA signal comes in to the left of U5 and exits to the right into the VGA header itself. The buffer is enabled via the chip enable signal CE (active low) at pin 19 which has STATUS_LED_BUS_MUX (P15) connected to it. The buffer chip itself is a high speed LVTTL 74LVC245A capable of signaling speeds in excess of those required to support a VGA signal that's 1600 pixels across (far in excess of what the Propeller can do). The data sheet for the 74LVC245A is located on the FTP site here:

PropC3 \ Docs \ Datasheets \ 74LVC245A.pdf

The signals for the VGA port are shared with Port B and listed once again in Table 2.9 below.

Table 2.9 — The VGA circuit signals

Signal Name	Propeller IO(Pin)	Description | Notes
VGA_RED_B1	P23 (26)	VGA Red Bit 1
VGA_RED_B0	P22 (25)	VGA Red Bit 0
VGA_GREEN_B1	P21 (24)	VGA Green Bit 1
VGA_GREEN_B0	P20 (23)	VGA Green Bit 0
VGA_BLUE_B1	P19 (22)	VGA Blue Bit 1
VGA_BLUE_B0	P18 (21)	VGA Blue Bit 0
VGA_HSYNC	P17 (20)	VGA HSync
VGA_VSYNC	P16 (19)	VGA VSync
STATUS_LED_BUS_MUX	P15 (16)	VGA buffer enable (active low).

2.8 Audio System

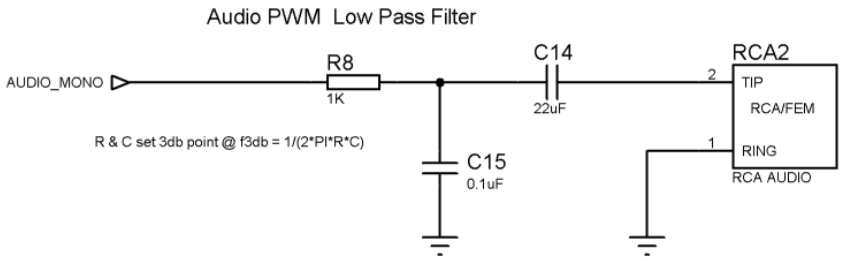
The Propeller chip has no dedicated sound hardware, so once again most designs either use some form of software generated sound or external sound hardware. The C3 uses the former concept and a single pin to generate sound using a PWM (pulse width modulated) analog signal to generate the signal. The concept of PWM is very powerful and if you haven't heard of it, or have, but don't understand it, it's worth your time to check it out. In brief, a single digital pin is toggled at a specific rate. The toggling of each cycle has a certain percentage of the signal high and a certain percentage low. The term for this high/low relationship is called **duty cycle**.

So, imagine if you had an analog integrator that could average the signal over time. If we toggle a 3.3 V signal with a duty cycle of 50%, then it stands to reason the output would be 3.3 V / 2 = 1.65 V. This is the general idea. We toggle a signal very fast with software/hardware and control the duty cycle. Therefore, we end up with a 1-bit D/A convertor that we can connect to a speaker and—presto—we have sound. The idea is that the PWM should be 10–100x faster than the signal you want to synthesize and the output of the PWM is voltage only, there is very little current. So, if you want to drive a speaker you need to add an op-amp or buffer to get some drive current.

The theory of PWM sound generation is rather complex, but you can find excellent resources on the web, and/or by studying some of the Propeller sound drivers themselves. Also, there is a very long and illustrative chapter on PWM sound generation in my book ***Game Programming for the Propeller Powered HYDRA***, available from Parallax. With that in mind, let's take a look at the schematic for the sound circuit shown in Figure 2.9 below. As usual, there is a high resolution copy of the schematic on the FTP site in:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_audio.png

Figure 2.9 — Propeller C3 audio generation circuit



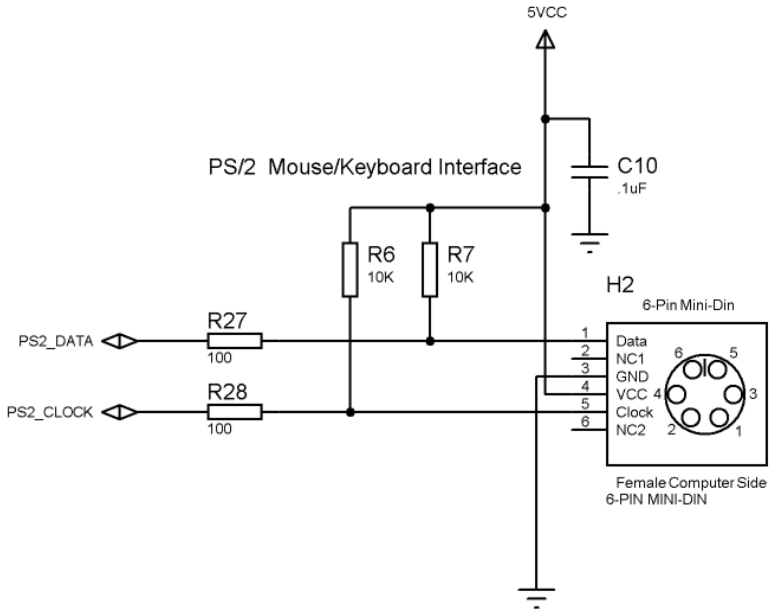
The circuit is driven by a single Propeller port P24 (31) connected to the system schematic signal AUDIO_MONO. A PWM signal drives the circuit which consists of a low-pass integrator filter made of R8 and C15. These components will integrate the PWM signal into a smooth DC voltage with the high frequency PWM riding on top of it. This is then removed by the second low-pass filter made of C14 and the impedance of the audio connection itself which is around 75 Ω if connected to an amplifier or TV set. The 22 µF capacitor starts kicking in at a few kHz, so it filters the high frequency PWM noise nicely, but allows very low (bass) to mid range audio signals to get through nicely with that 22 µF capacitor.

2.9 PS/2 Keyboard/Mouse Port

The PS/2 port on the Propeller C3 uses a standard DIN 6 socket. PS/2 protocol is based on a serial data packet that is clocked at a specific rate. Thus, whenever there is a key press or mouse movement (depending on what you have plugged in) a data packet(s) is sent. Usually, 11-bit packets consist of a start bit, 8 data bits, stop bit, and parity. Electrically, the PS/2 port requires +5 V, a clock line with a pull up, and a data line with a pull up. Then the majority of work is in the keyboard/mouse drivers. The schematic for the Propeller C3 PS/2 port is shown below in Figure 2.10. A high resolution copy of the circuit is located on the FTP site here:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_ps2.png

Figure 2.10 — PS/2 circuit on the Propeller C3.



The PS/2 interface is rather straightforward. **PS2_DATA** and **PS2_CLOCK** are connected to IO pins (P26, P27) on the Propeller and used for and data and clock respectively. The Propeller acts as the master controlling the clock signal while the keyboard toggles the data line when sending to the Propeller. Of course, the Propeller can send packets to the keyboard or mouse as well. This is why it's important that the circuits are open collector and why the pull ups **R6** and **R7** are required. The PS/2 spec states that the keyboard and mouse only has to pull the data and clock line to ground, not to drive it high; it should be pulled up.

Finally, **R27** and **R28** are current limiters to protect the Propeller from the +5 TTL signaling coming from the keyboard or mouse. Luckily, CMOS 3.3 V can drive 5 V TTL, and 5 V TTL can drive 3.3 V CMOS with a current limiter, so all is well. There are a number of PS/2 device drivers for mice, keyboards, and other PS/2 devices (since they are serial in nature). But, if you want to write your own driver there are numerous articles and white papers on the internet and there is a whole chapter on writing mouse and keyboard drivers in my book ***Game Programming for the Propeller Powered HYDRA*** which is available from Parallax. That's about all there is to the PS/2 circuit, Table 2.10 below lists the signals at a glance.

Table 2.10 — The PS/2 keyboard/mouse interface circuit signals.

Signal Name	Propeller IO(Pin)	Description | Notes
PS2_DATA	P26 (33)	PS/2 serial data non-inverted.
PS2_CLOCK	P27(34)	PS/2 serial clock non-inverted.

2.10 FLASH Memory System

The FLASH memory on the Propeller C3 is comprised of a single 8 Megabit/1MByte SPI FLASH chip from Atmel part **AT26DF081A**. The chip can signal in excess of 25 MHz, so its plenty fast to use as a large storage area for assets, code, data, images, whatever. Moreover, FLASH memories are very easy to read and write from since they are sector based and have small state machines inside them to make life easy on you. Additionally, there are many read/write modes that help you optimize access to the FLASH memory. Later in the software section, we will see a demo of how to read and write to the FLASH memory, but you will need to read the data sheet at some point. You can find it on the FTP site here:

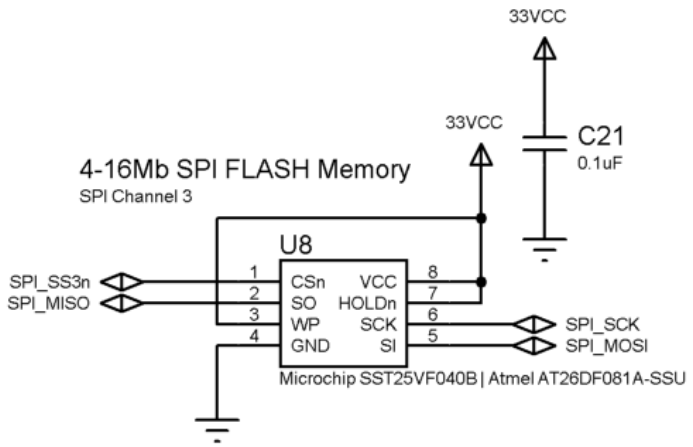
PropC3 \ Docs \ Datasheets \ AT26DF081A.pdf

Now, at some point the C3 might use another FLASH memory. The good news is that 95% of FLASH memories have the same instruction set and features. But, there are differences between parts even though they are supposed to be JEDEC standardized. Thus, if a C3 is shipped with a variation on the FLASH memory that breaks some of the driver or demo code, we will let you know and provide updated drivers. That said, for the most part if you have programmed one FLASH memory, you have programmed them all. Read the data sheet carefully; if you want to utilize the FLASH memory to its fullest on the C3, but in short, you power it up, unprotect it, then start reading or writing single bytes in sequential access mode. It's that simple.

The schematic for the FLASH memory circuit is nothing more than the FLASH memory itself as shown in Figure 2.11 below. Also, there is a high resolution copy of the schematic located in the FTP site here:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_flash.png

Figure 2.11 — FLASH memory circuit on Propeller C3.



The FLASH chip is labeled **U8** and is connected to the system SPI bus signals (SPI_MOSI, SPI_MISO, and SPI_SCK) which are common to all SPI devices on the bus. Then the chip select signal for the FLASH memory is connected to pin 1 (**CSn**) which in this case is **SPI_SS3n** coming from the SPI bus logic. Thus, the FLASH is channel 3 on the SPI bus. Therefore, if you want to access the FLASH memory on the SPI bus, you reset the SPI counter, then clock it to channel 3, and the chip select for the FLASH memory will assert and you are free to communicate with the FLASH memory. Also, note that **HOLDn** and **WP** (write protect) are tied high. Table 2.11 below lists the signals at a glance for the FLASH memory.

Table 2.11 — FLASH SPI signals.

Signal Name	Propeller IO(Pin)	Description | Notes
SPI_MOSI	P9 (10)	SPI master out slave in
SPI_MISO	P10 (11)	SPI master in slave out
SPI_SCK	P11 (12)	SPI clock
SPI_SS3n	NA (internal)	Selects 1MB FLASH memory

2.11 32K x 2 SRAM Design

The SRAMS on the Propeller C3 are one of the most interesting design aspects. As discussed in the overview about the design on the C3, in a perfect world, we would like to have parallel static SRAMs connected to the Propeller. However, with the large number of peripherals the C3 is incorporates, there simply isn't enough IO to go around and support a parallel SRAM system without a lot of expense. Therefore, a compromise was to use new SPI based serial SRAMs. These are VERY fast, 25 MHz and greater and since we are dealing with microcontroller speeds here on the order of 10-20 MIPS for the Propeller on average for 1-2 clocks per instruction, 25 MHz is more than enough to feed even the most demanding applications.

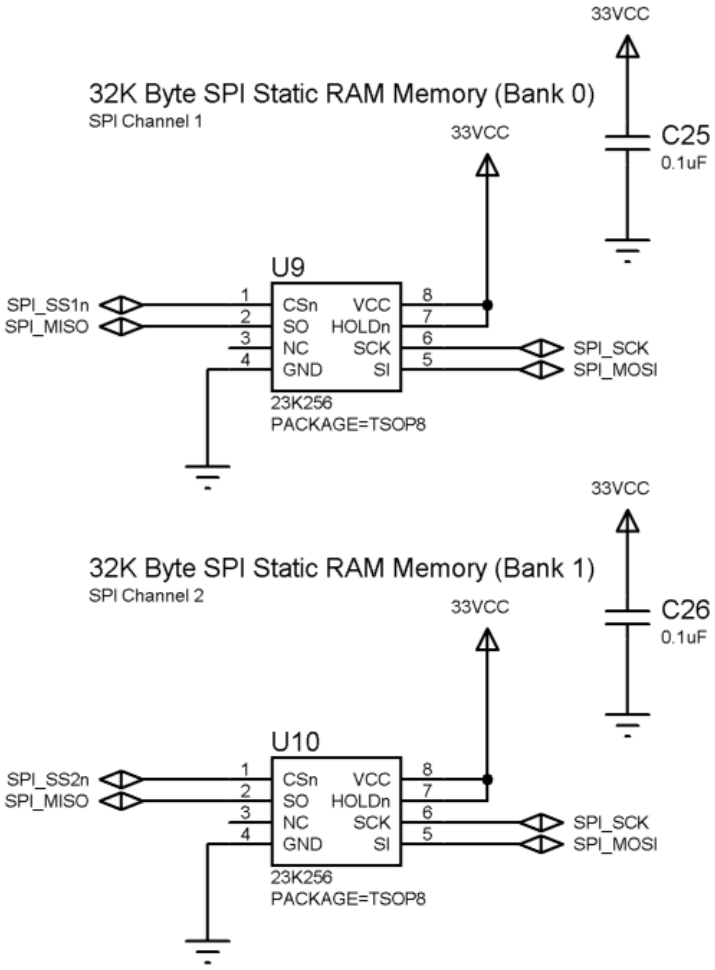
For example, if you assume you have to clock in a byte with 8 bits, plus 2 bits of overhead, you get a 2.5 MHz memory bandwidth. Even if you cut that in half to be safe, that's 1.25 Mbytes/sec. And that's not too shabby! Considering old 8-bit PCs had about 1/2 to 1/10th that bandwidth, they did just fine. In my searches, I found that Microchip is the leader in serial SPI SRAMs and have the largest one available: 32 K bytes. This is huge for a serial SRAM. So, I decided to put a pair of them on the C3 for a total of 64 KB. You can access each one on the SPI bus, as if they were any other SPI device. The data sheet for the device is located on the FTP site here:

PropC3 \ Docs \ Datasheets \ 23K256.pdf

The schematic for the SRAM memory circuit is nothing more than a pair of SPI SRAMs as shown in Figure 2.12 below. Also, there is a high resolution copy of the schematic located in the FTP site here:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_sram.png

Figure 2.12 — Dual 32 KB SRAM memory circuit on Propeller C3.



The SPI SRAMs **U9** and **U10** are connected to the system SPI bus signals (SPI_MOSI, SPI_MISO, SPI_SCK). The chip selects for the SPI SRAMs use **SPI_SS1n** (channel 1) and **SPI_SS2n** (channel 2) for bank 0 (U9) and bank 1 (U10) respectively. To access either SRAM, simply assert channel 1 or 2 and then the SRAM is yours to command. Interestingly, they are very similar to the FLASH memory, expect that there is no erasing necessary and it's not sector based like the FLASH. The SRAMs are completely random access for both reading and writing. Also, one note about optimization… I decided to place the SRAMs at SPI channel 1,2 since in most cases accessing SRAM is more frequent that FLASH or ROM memories, thus after an SPI reset, all you need to do is clock the SPI counter a single time, and you are accessing SRAM bank 0. That wraps it up for the SRAMs. Table 2.12 below lists the signals at a glance for the SRAM circuit.

SRAM on power-up — When the SRAMs power up they will have random data in them, so be sure to clear them out and don't assume they have 0's in them at boot. Also, the SRAMs have small 0.1 µF bypass/filter capacitors on them (C25, C26) and due to their extremely low power consumption they will actually hold state for a few seconds being powered by the small capacitor! This is a cool feature if there is a momentary power glitch, but regardless, you should always clear any SRAM variables or storage before using them.

Table 2.12 — SRAM SPI signals.

Signal Name	Propeller IO(Pin)	Description | Notes
SPI_MOSI	P9 (10)	SPI master out slave in
SPI_MISO	P10 (11)	SPI master in slave out
SPI_SCK	P11 (12)	SPI clock
SPI_SS1n	NA (internal)	Selects SRAM bank 0
SPI_SS2n	NA (internal)	Selects SRAM bank 1

2.12 A/D System

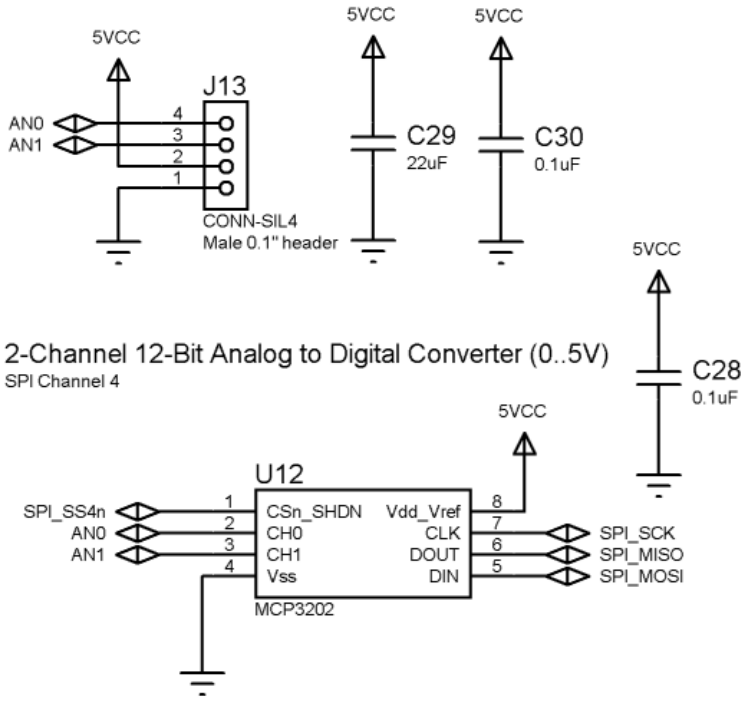
Once I had the SPI bus system in place on the Propeller C3, it was tempting to add all kinds of devices to the bus (and it still is!). However, we wanted to keep cost down, so I made a list of what people really use and A/D and D/A came up over and over. Since, A/D is obviously the most complex process, that's what I decided to add. Now, if you're a Propeller programmer, you know that there are software A/D drivers, but they are slow, plus they use up one or more processors for the conversion. What we needed was a real A/D with 1-2 channels and 10-12 bits of resolution that had an SPI interface. After searching a while I came upon the MCP3202 from Microchip, I have used this device and its relatives many times and found it to be very reliable, fast, low cost, and easy to program. Additionally, I searched Parallax's website and forums and found that many customers have used the MCP3202 and there are even Propeller drivers for it, thus, the decision was easy to make for the MCP3202. It's a dual channel 12-bit, successive approximation A/D that supports either 2 ground referenced analog inputs or a single differential channel with a conversion rate of 100 K samples per second (at +5.0 V reference). The data sheet for the MCP3202 device is located on the FTP site here:

PropC3 \ Docs \ Datasheets \ MCP3202.pdf

The schematic for the A/D circuit includes both the header itself as well as the MCP3202 as shown in Figure 2.12 below. Also, there is a high resolution copy of the schematic located in the FTP site here:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_ad.png

Figure 2.13 — MCP3202 Analog to Digital converter circuit on the Propeller C3.



Let's start with the header itself **J13**; this is where you will inject your analog signals for sampling. First, you will notice that there are two bypass capacitors **C29** (22 µF) and **C30** (0.1 µF). The larger capacitor **C29** is for power smoothing, and the small capacitor **C30** is for high-frequency noise filtering. Both are needed to keep the power from the header as clean as possible, since you may be using the +5.0 V from the header to power your analog device generating the signals for sampling. If you don't use the power from the header then you at least need to connect the ground pin at Pin 1 of the header (bottom) to your signal references ground. Also, the MCP3202 uses +5.0 V as its reference voltage; therefore, any signals you inject for sampling can range from 0..5 V. If you need a smaller range that's ok, you aren't going to hurt the chip, but if you need a larger range then I suggest a voltage divider or scaler.

Moving onto the MCP3202 itself, once again, it's very simple since it's an SPI device. As usual, the SPI bus is connected to the device with signals (SPI_MOSI, SPI_MISO, SPI_SCK) and this time the SPI channel used by the A/D converter is **SPI_SS4n** (channel 4). To access the MCP3202, simply advance the SPI selection logic to channel 4 and start sending commands to the A/D. Table 2.13 below lists the signals for the MCP3202 and SPI interface.

The MCP3202 data sheet shows the input model of each channel. Each is composed of a number of components and a capacitor to sample the signal. One thing that is important is that the output impedance of your source is low enough that the MCP3202 can sample it. Therefore, read the data sheet on this topic, but as long as you have impedances in the 1 kΩ–100 kΩ range it should be fine. For example, if you hook a potentiometer across +5 and ground, with the wiper as the signal source, 1,10..100 kΩ will work fine, if you use a 1 MΩ potentiometer you will start to notice more noise and less accuracy.

Table 2.13 — MCP3202 Analog to Digital converter SPI signals

Signal Name	Propeller IO(Pin)	Description | Notes
SPI_MOSI	P9 (10)	SPI master out slave in
SPI_MISO	P10 (11)	SPI master in slave out
SPI_SCK	P11 (12)	SPI clock
SPI_SS4n	NA (internal)	Selects A/D converter
J13 Analog Input Port		
AN0	NA (J13 pin 4)	Analog input 0 (0..5 V)
AN1	NA (J13 pin 3)	Analog input 1 (0..5 V)
+5V	NA (J13 pin 2)	Power from C3
GND	NA (J13 pin 1)	Ground from C3 directly to A/D

2.13 Secure Digital (SD) Card Interface

The last SPI device accessible on the Propeller C3 is of course the SD card itself. In reality, SD cards are not primarily SPI devices. The SPI mode of operation was a fallback or slow mode designed in when IO was at a premium. SD cards have a much faster parallel mode that utilizes a 4-bit bus. However, this parallel mode requires a license payment each year, thus only large companies use it. The SPI mode is free to use and still supports the full functionality of the SD card protocol. With that said, SD cards are very complex devices and similar to IDE disk drives in many ways at least as far as their complexity goes. If you are interested in learning how to write a driver there are numerous documents online including the SD specification itself. You can find copies of the documents on the FTP site here:

PropC3 \ Docs \ SD \ *.*

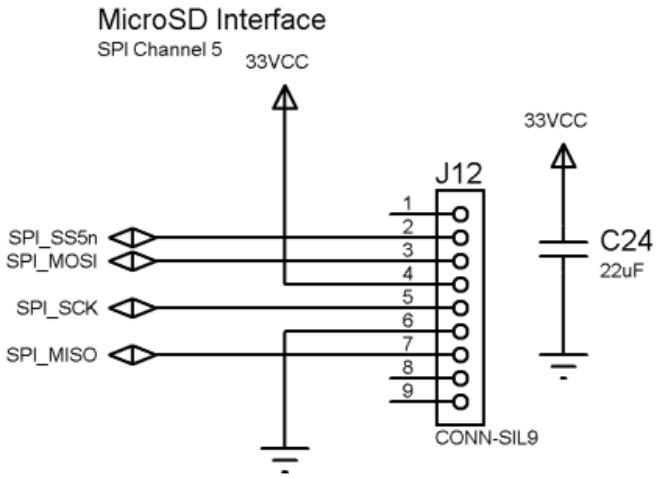
There is documentation for SD specification, FAT, and others; I suggest reading them all if you want to write your own driver. And speaking of drivers for SD cards and the Propeller chip, there aren't a lot of them (due to their complexity); however, I have written one for you to get started (which you will learn about in the software section). Also, I wrote a very detailed document on SD card driver development for an SD card add-on for the HYDRA game console. The document is of course specific to the HYDRA SD Max product and refers to it, but the theoretical section on SD cards, FAT, and writing drivers is the most complete you will find. I know since I looked for months! And that's why I had to write it myself. The document and any other ancillary information can be found here on the FTP site:

PropC3 \ Docs \ HYDRASDMAX \ *.*

With that in mind, let's look at the schematics for the SD card interface (which is nothing more than a mechanical interface actually, there are no electronics, it's all in the SD card itself). Figure 2.14 below shows the SD card interface. Also, there is a high resolution copy of the schematic located in the FTP site here:

PropC3 \ Designs \ Schematics \ prop_c3_rev_a_sd.png

Figure 2.14 — The SD card interface schematic.



As you can see, there isn't much to the SD card header electrically. It uses the SPI bus as usual and, the SD card allocates channel 5 for its SPI select. The one thing that is important is the use of the large 22 µF capacitor on the device. SD cards tend to draw a lot of current when they mount/boot, the 22 µF capacitor helps stabilize the power. I have seen designs that had trouble mounting SD cards due to 100's of milliamps being pulled momentarily that the designer didn't account for. This in turn causes a drop in the power supply voltage and browns out the SD card. So, it's always good to put a nice 22–33 µF capacitor on your SD cards in addition to a 0.1 µF bypass capacitor. Table 2.14 below lists the signals that the SD card interface uses.

Table 2.14 — SD card SPI signals.

Signal Name	Propeller IO(Pin)	Description | Notes
SPI_MOSI	P9 (10)	SPI master out slave in
SPI_MISO	P10 (11)	SPI master in slave out
SPI_SCK	P11 (12)	SPI clock
SPI_SS5n	NA (internal)	Selects microSD card

2.14 Adding SPI Devices to the Propeller C3

The Propeller C3 has two extra SPI select channels 6,7 exported out to the SPI / I^2C header along with (SPI_MOSI, SPI_MISO, SPI_SCK). Table 2.15 below lists the header signals for reference.

Table 2.15 — Signals top to bottom for SPI / I^2C Header.

Signal Name	Propeller IO (Pin)	Description
SS7	NA (generated internally)	SPI select channel 7 (active low)
SS6	NA (generated internally)	SPI select channel 6 (active low)
SCLK	P11 (12)	SPI clock
MISO	P10 (11)	SPI master in slave out (into Propeller)
MOSI	P9 (10)	SPI master out slave in (from Propeller)
PSCL	P28 (35)	Serial I^2C clock out of Propeller
PSDA	P29 (36)	Serial I^2C data in/out of Propeller
GND	Ground	System ground

Adding another SPI device is as easy as connecting the header bus signals labeled MOSI, MISO, SCLK to your SPI device, along with power (3.3 V, GND) and finally the SPI select signal SS6 or SS7. Be careful that you put proper bypass capacitors on your SPI device and it's a 3.3 V device. If it's a 5 V device, you can still use it, but you will have to put a voltage translator on it or voltage divider at least on the MISO signal to make sure it doesn't damage the Propeller chip.

SPI Signal Output — The SPI signals to the header itself actually have 100 Ω resistors inline to help protect from over current/voltage, but they will require a 3.9 kΩ or higher resistor in series to be 5 V TTL compatible, but a better idea is to reduce any input voltage to the Propeller to 3.3 V at max.

3 Demos and API

The Propeller C3 is like any other Propeller platform in respect to NTSC, PAL, VGA, PS/2, serial communications, sound, etc. Any driver written for a Propeller chip that utilizes the standard design patterns on the currently manufactured Propeller development boards work on the C3 with little or no change other than an IO pin assignment. Nonetheless, putting together a quick NTSC demo or PS/2 keyboard test takes only a few lines of code or a page or two at most (of course, writing about each takes hours!) That said, to be complete the following demos show off all the systems of the Propeller C3 for those readers who are new to the Propeller chip.

3.1 What to Expect

Each demo/test is rather simple and the bare minimum to get you going. Many of them rely on pre-written drivers from the **Parallax Object Exchange**, myself, or others. However, the SRAM, FLASH, A/D, and SD card demos I wrote specifically in Spin from the ground up and made them very easy to understand with generous comments. So, if you are interested in using the SRAM for example, all you need to do is read the data sheet, then try out the demo, read the source, copy the functions you are interested in and that's that. Additionally, if I developed an API for the device in question (even if it's 2-3 functions), I will discuss the API in the section and list the important functions for you at a glance.

Initially, I thought to turn everything into driver objects… but decided that I don't want to try to anticipate what everyone wants, and better to just show a simple example and let the user (you) take what you want. The only demo that relies on more complex software and an actual driver (still in Spin) is the SD card demo. This is a complex beast no matter how you slice it!

Also, there seems to be a lot of interest currently in communicating with Propeller boards over a serial link using the **Parallax Serial Terminal** which you can download from this link:

http://www.parallax.com/Portals/0/Downloads/sw/propeller/Parallax-Serial-Terminal.exe

Therefore, for key demos, I ported them to use serial communications in addition to the "local" PS/2 keyboard versions—which brings me to the requirements of all the demos. You will need the following to try all of them out:

- Windows PC with the Propeller Tool and a USB A to Mini-B cable
- VGA monitor for the VGA demo
- NTSC TV / Monitor for most demos, since I used an NTSC video driver for video
- PS/2 keyboard for most demos since I use it as an input device for menus
- PS/2 mouse for mouse demo
- Parallax Standard servo or compatible part #900-00005 (for servo demo)
- Parallax NES Gamepad Controller Adapter board Part #32368 along with an NES controller
- MicroSD card formatted FAT16 with some text files on it

Of course, many of these devices are optional and only needed if you want to play with the associated feature. For example; NES controllers to try the NES demo, servos to try the servo demo, etc.

3.2 System Setup for the Tests and Demos

Each demo will have specific requirements, but let's get the C3 set up and everything ready. Here's a list of bare minimum things to setup:

- First, we are obviously going to use the Propeller Tool, so launch that and hook your USB cable to the C3.
- Most of the demos require an NTSC display (audio is nice too), so hook up an NTSC TV.
- All of the "local" demos require a PS/2 keyboard.
- The VGA monitor is only required for the VGA demo.
- All the software for the demos can be found on the FTP site here:

  PropC3 \ Sources \ *.*

At this point, you need to copy the **Sources** directory and all sub-directories to your hard drive, so the demos hereafter can find all the files properly. In fact, I suggest you simply drag the entire contents of **PropC3** to your hard drive.

I will assume you know your way around the Propeller Tool, if not, read the online documentation or review the **Quick-Start Guide** in this manual. Each demo will follow the same pattern more or less: overview, load the software, set up the C3, try the demo out, brief discussion and review any API used in the demo. Without further ado, let's get started!

3.3 Local Version Demos (PS/2 + NTSC Monitor)

This section contains the **"local"** version test demos. By "local" version, we mean that the C3 is controlled by a local PS/2 keyboard plugged into the C3, so you type commands into it and see the results on the NTSC monitor, or the movement of a servo or the specific IO device being tested. The point is you need an NTSC TV hooked up to the A/V port of the C3 along with a PS/2 keyboard.

3.3.1 Keyboard & Mouse Demo

The keyboard/mouse demo uses standard Parallax PS/2 keyboard and mouse drivers to communicate with the C3 PS/2 port. The top level file for the demo is named **c3_keymouse_demo_010.spin** and you can find it on the FTP site here:

PropC3 \ Sources \ c3_keymouse_demo_010.spin

The demo includes a number of other objects for video, keyboard, mouse, etc. so make sure you drag the entire Sources\ directory to your hard drive, so the Propeller Tool can find them. The demo allows you to plug in either a keyboard or a mouse and try them out. Figure 3.1 below shows screen shots of the demo in action.

Figure 3.1 — The Keyboard/Mouse demo in action.

(a) The main menu. (b) The keyboard demo in action. (c) The mouse demo in action.

Referring to Figure 3.1 (a), (b), (c) from left to right, we see a simple menu that allows you to select keyboard or mouse. Now, ironically, you must have the keyboard plugged in to select which menu option you want to try; keyboard or mouse! But, once you select mouse, you will be instructed to insert the mouse and the software will begin the demo once it detects a mouse has been inserted.

The demo relies on the following objects as shown in the code fragment below:

```
OBJ

' current drivers used in this version of SPI interface
kbd           : "keyboard_010.spin"                 ' instantiate keyboard driver
mouse         : "mouse_010.spin"                    ' instantiate mouse driver
gfx_ntsc      : "C3_GFX_DRV_010.spin"               ' instantiate new NTSC tile driver
```

The keyboard and mouse drivers are standard Parallax objects that are system agnostic. The GFX_NTSC driver is a tile engine I developed that has 32x25 lines of text with color control, scrolling, and other features for games and graphics. It's used as a stock driver in many of the demos, so we can see something on the NTSC screen. It could be easily replaced by any of 100 graphics drivers for the Propeller, but I like it since its small, fast, low memory imprint and I wrote a complete terminal emulator in it as well, so it is very easy to "printf" to it and do console based applications in color.

In any event, if you're interested in the graphics driver, just read the source for it; the documentation is built into the source code. Moving on, reading the keyboard and mouse on the C3 is very simple, all you need is the IO port pins, and then you call the Parallax driver(s) as shown below for the keyboard (mouse is similar):

```
    'start keyboard on c3, keyboard is always started even if its not plugged in
    ' we need it to read the user selection :)
    kbd.start(PS2_DATA, PS2_CLK)
```

The keyboard and mouse drivers both have a lot of functionality, so I suggest reading the source code if you are interested in seeing their complete API. Here's the mouse demo fragment from the program for reference:

```
PUB Mouse_Demo

  '////////////////////////////////////////////////////////////////////////////
  ' Mouse TEST SUITE ////////////////////////////////////////////////////////
  '////////////////////////////////////////////////////////////////////////////
  {{
  This test simply echos the keyboard to the NTSC screen. Make sure PS/2 keyboard
  is plugged into C3 and C3 is plugged into NTSC.
  }}
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string (''C3 Mouse Test.'') )
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string (''- Prints x,y coords to screen.''))
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string (''- Plug in PS/2 Mouse...'') )
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string (''- Press either button to exit.'') )
  gfx_ntsc.Newline_Term

  'start mouse on c3 pins
  mouse.start(PS2_DATA, PS2_CLK)

  ' wait for mouse to be inserted

  repeat while (g_temp1 := mouse.present) == 0

  ''   3 = five-button scrollwheel mouse
  ''   2 = three-button scrollwheel mouse
  ''   1 = two-button or three-button mouse
  ''   0 = no mouse connected

  gfx_ntsc.Dec_Term( lookup ( g_temp1: 2,3,5 ) )
  gfx_ntsc.String_Term( string ('' button mouse detected.'') )
  gfx_ntsc.Newline_Term

  ' delay a moment
  Delay_US(1*1_000_000)

  ' print mouse relative deltas to screen
  repeat

      ' test for mouse button clock
      if (mouse.buttons > 0)
        ' unload mouse driver
        mouse.stop
        return

      ' print x,y deltas to screen
      gfx_ntsc.String_Term( string (''Mouse(x,y) = (''))
```

```
      gfx_ntsc.Dec_Term( mouse.abs_x )
      gfx_ntsc.String_Term( string (",") )
      gfx_ntsc.Dec_Term( mouse.abs_y )
      gfx_ntsc.String_Term( string (")") )
      gfx_ntsc.Newline_Term

' end Mouse_Demo
```

The function starts by installing the mouse driver, waiting for the mouse to be installed and then begining the demo that prints out the mouse position and button state to the screen with the graphics driver's terminal functionality.

3.3.2 VGA Demo

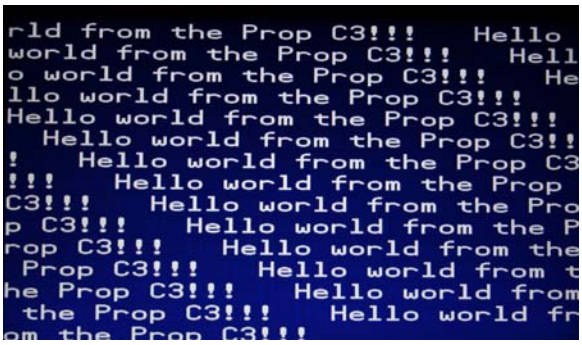
The VGA demo uses a standard Parallax VGA terminal driver **VGA_Text_010.spin** to print to the VGA screen **"Hello world from the Prop C3!!!"**. The C3 uses the same VGA IO pins as the Propeller Demo board and HYDRA **(P23..P16)**, so there are no changes required to the driver. Otherwise, you would have to go into the code and find the pin group and change it to the C3's VGA pin group **P23..P16**. Other than that, all you need is a VGA monitor plugged into the C3's VGA port.

The top level file for the demo is named **c3_vga_demo_010.spin** and you can find it on the FTP site here:

PropC3 \ Sources \ c3_vga_demo_010.spin

The demo doesn't do much other than print "Hello world from the Prop C3!!!" to the VGA screen. Figure 3.2 below shows screen shots of the demo in action.

Figure 3.2 — Screen shot of VGA demo running.



The code for the demo is ridiculously simple; below is a fragment with the **OBJ** and main **PUB** sections including the main function, so we can discuss it briefly (highlighted code in white):

```
'/////////////////////////////////////////////////////////////////////////////
'OBJS SECTION ////////////////////////////////////////////////////////////////
'/////////////////////////////////////////////////////////////////////////////

OBJ

' current drivers used in this version of SPI interface
term_vga   : "VGA_Text_010.spin"    ' instantiate VGA terminal driver

'/////////////////////////////////////////////////////////////////////////////
' MAIN ENTRY POINT TO DEMO/////////////////////////////////////////////////////
'/////////////////////////////////////////////////////////////////////////////
```

```
PUB Main : status

  ' let the system initialize
  Delay_US( 1*1_000_000 )

  ' initialize the vga driver
  ' take note of the control bits for this particular driver -  %10111
  term_vga.start(%10111)

  ' delay a moment
  Delay_US(1*1_000_000)

  ' ////////////////////////////////////////////////////////////////////////
  ' INITIALIZE I/O PINS ////////////////////////////////////////////////////
  ' ////////////////////////////////////////////////////////////////////////

  ' blink the status LED 3 times to show board is "alive", then enable VGA (leave LOW)
  DIRA[STATUS_LED_BUS_MUX] := 1 ' set to output
  OUTA[STATUS_LED_BUS_MUX] := 0

  repeat 6
    OUTA[STATUS_LED_BUS_MUX] := !OUTA[STATUS_LED_BUS_MUX]
    repeat 125_000

  ' turn on VGA buffer (and status LED)
  OUTA[STATUS_LED_BUS_MUX] := 0

  ' initialize SPI IO (eventhough, we aren't using it in this demo at all,
  ' good habit to set it up)
  SPI_Init

  ' and here's the demo!
  term_vga.Out( $00 )

  repeat
    term_vga.pstring( string ("Hello world from the Prop C3!!!   ") )
    Delay_MS(75)

' end Main
```

The single VGA driver object is included at the top in the **OBJ** section, then we fall into the **Main()** function. The first and most important thing that happens here is we start the VGA driver with the correct control bits. Refer to the driver itself and the **start()** function for more details, but the **%10111** code sets things up correctly for the C3. Next, we get into the setting of the IO port **STATUS_LED_BUS_MUX**; remember this controls the VGA buffer and we have to enable it (active **low**) for VGA to make it out to the HD15 header. That's what the highlighted code does.

Finally, we make to the main loop of the program which simply prints to the screen the "Hello world..." message in an infinite loop or at least until our sun goes super nova.

3.3.3 Audio Demo

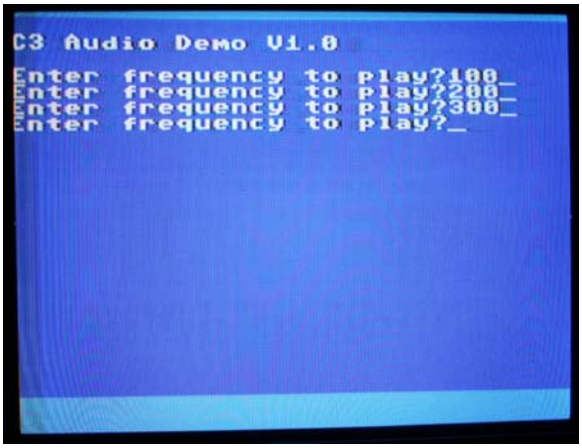
The audio demo uses a single Propeller IO port routed into a low-pass filter. This is how most Propeller boards generate sound. As discussed in the Hardware portion of this manual, PWM signals are used to generate an analog signal. This is how the C3 works as well, so any audio driver that uses PWM and sends it out a Propeller IO pin will work to generate sound. You just have to set the pin correctly on the driver which on the C3 is named **AUDIO_MONO** (P24 on the C3). For our little demo, we are going to use a sound driver developed for the HYDRA, named **NS_sound_drv_052_11khz_16bit.spin**. The

driver is very powerful and its documentation is embedded in the source comments. The demo program's top level file is named **c3_audio_demo_010.spin** and you can find it on the FTP site here:

PropC3 \ Sources \ c3_audio_demo_010.spin

The demo prompts the user to enter in a frequency in Hz and then plays the sound in a chord for a couple seconds with a specific ADSR envelope. Figure 3.3 below shows screen shots of the demo in action. Of course, you MUST have an NTSC TV with A/V hooked up to the C3 along with a PS/2 keyboard to enter data.

Figure 3.3 — Screen shot of the audio demo in action.



Referring to the figure, all you do is enter in a frequency (0..3000 Hz) and the demo plays the sound for a couple seconds. The actual code that generates the sound makes a call to a function called **PlaySoundFM()** as shown in the fragment below (note that the left-arrow symbol "←"indicates that the line of code wraps due to space limitations in print, but it should be on one line when used.):

```
   ' play the sound
   snd.PlaySoundFM(0, snd#SHAPE_SQUARE, g_temp1, CONSTANT( ←
Round(Float(snd#SAMPLE_RATE) * 2.8)), 200, $3579_ADEF )

   snd.PlaySoundFM(1, snd#SHAPE_SQUARE, (g_temp1*5)/4, CONSTANT( ←
Round(Float(snd#SAMPLE_RATE) * 2.9)), 200, $3579_ADEF )

   snd.PlaySoundFM(2, snd#SHAPE_SQUARE, (g_temp1*6)/4, CONSTANT(←
Round(Float(snd#SAMPLE_RATE) * 3.0)), 200, $3579_ADEF )
```

As you can see, we make 3 calls to the sound function which starts 3 independent sounds at the same time, thus creating a chord effect. We modify the frequency of each taking into consideration how chords are supposed to be generated along with a little artistic license to make them sound good.

You can learn more about **PlaySoundFM()** by reviewing the sound driver itself, but here's the main API calling information for reference:

```
PUB PlaySoundFM(arg_channel,arg_shape,arg_freq,arg_duration,arg_volume,arg_amp_env)|offset
{{
Starts playing a frequency modulation sound. If a sound is already
playing, then the old sound stops and the new sound is played.

   arg_channel:   The channel on which to play the sound (0-5)
   arg_shape:     The desired shape of the sound. Use any of the
                  following constants: SHAPE_SINE, SHAPE_SAWTOOTH,
                  SHAPE_SQUARE, SHAPE_TRIANGLE, SHAPE_NOISE.
```

```
                    Do NOT send a SHAPE_PCM_* constant, use PlaySoundPCM() instead.
    arg_freq:       The desired sound frequncy. Can be a number or a NOTE_* constant.
                    A value of 0 leaves the frequency unchanged.
    arg_duration:   Either a 31-bit duration to play sound for a specific length
                    of time, or (DURATION_INFINITE | "31-bit duration of amplitude
                    envelope") to play until StopSound, ReleaseSound or another call
                    to PlaySound is called. See "Explanation of Envelopes and
                    Duration" for important details.
    arg_volume:     The desired volume (1-255). A value of 0 leaves the volume unchanged.
    arg_amp_env:    The amplitude envelope, specified as eight 4-bit nybbles
                    from $0 (0% of arg_volume, no sound) to $F (100% of arg_volume,
                    full volume), to be applied least significant nybble first and
                    most significant nybble last. Or, use NO_ENVELOPE to not use an envelope.
                    See "Explanation of Envelopes and Duration" for important details.
}}
```

3.3.4 Port A/B IO Demo

The port IO demo is nothing more than a few lines of code that takes user input and then writes to the IO port headers A (P7..P0) or B (P23..P16) or reads the ports and prints to the NTSC screen. Not really worth a demo, but the program is a nice template if you want to bit-bang the IO ports and read / write from a console application running on the C3. The top level file for the demo is named **c3_port_io_demo_010.spin** and you can find it on the FTP site here:

PropC3 \ Sources \ c3_port_io_demo_010.spin

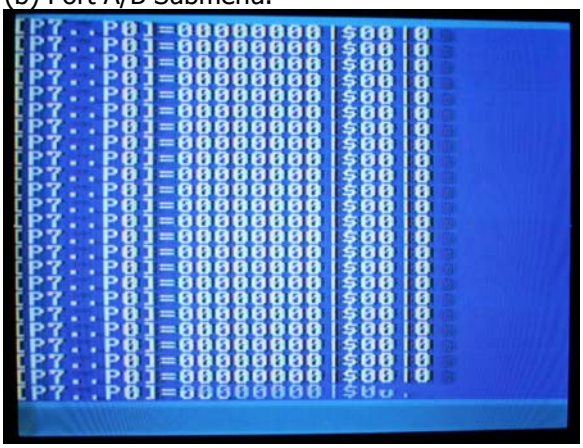
Figure 3.4 below is a screen shot of the demo running. It has a little menu that allows you to select port A or B, then a sub-menu that prompts for read or write to the port. As usual, you will need the NTSC monitor and PS/2 keyboard plugged in.

The input parser for these demos allows you to enter data in decimal (default), hex $xx, or binary %xxxxxxxx. Thus, for the bit-banging experiments you wanted to write all 1's to the lower nibble of port A and all 0's to the upper nibble. This would be 15 in decimal, but a more direct representation, so you don't have to convert your binary to decimal would be to write %00001111 as the input value. The parser is smart enough to convert. Just make sure you prefix hex numbers with "$" and binary numbers with "%".

Figure 3.4 — Screen shots of port IO demo.

(a) Main menu.

(b) Port A/B Submenu.



As an example, below is a fragment of the entire function that processes IO Port B. It is nearly identical to the Port A code except that Port B (**P23..P16**) shares its IO with the VGA port, thus, the VGA buffer has to be enabled, disabled as the port is used for general IO.

```
PUB Port_B_Test

  '////////////////////////////////////////////////////////////////////////////
  ' VGA PORT B (P24..P16) "IO" TEST SUITE ////////////////////////////////////
  '////////////////////////////////////////////////////////////////////////////

  ' simply allows user to read/write values to port IOs P24..P16 shared with VGA

  repeat
    ' draw menu
    gfx_ntsc.Newline_Term
    gfx_ntsc.String_Term( string ("VGA Port B IO Test Menu") )
    gfx_ntsc.Newline_Term
    gfx_ntsc.Newline_Term
    gfx_ntsc.String_Term( string ("1. Read Port IOs P23..P16.") )
    gfx_ntsc.Newline_Term
    gfx_ntsc.String_Term( string ("2. Write Port IOs P23..P16.") )
    gfx_ntsc.Newline_Term
    gfx_ntsc.String_Term( string ("3. Exit back to main menu.") )
    gfx_ntsc.Newline_Term
    gfx_ntsc.Newline_Term
    gfx_ntsc.String_Term( string ("Selection?") )

    ' get string from user
    Get_String(@g_sbuffer, 9 )

    ' convert to integer
    g_key := atoi2(@g_sbuffer, 9)

   ' what is user requesting?
   case g_key
     1:
       ' make sure VGA is disabled (0 = ENABLED, 1=DISABLED)
       OUTA[STATUS_LED_BUS_MUX] := 1

       ' set VGA shared IOs to inputs
       DIRA[23..16]             := %00000000

       gfx_ntsc.Newline_Term
       gfx_ntsc.Newline_Term
       gfx_ntsc.String_Term( string ("-Press any key to stop scan-") )
       repeat 500_000

       ' scan and print IOs
       repeat while (kbd.gotkey == FALSE)
         gfx_ntsc.Newline_Term
         gfx_ntsc.String_Term( string ("[P23..P16]=") )
         gfx_ntsc.Bin_Term( INA[ 23..16 ], 8 )
         gfx_ntsc.String_Term( string("|$"))
         gfx_ntsc.Hex_Term( INA[ 23..16 ], 2 )
         gfx_ntsc.Out_Term("|")
         gfx_ntsc.Dec_Term( INA[ 23..16 ] )

       kbd.key ' flush key
       gfx_ntsc.Newline_Term
       gfx_ntsc.Newline_Term

     2:
       ' make sure VGA is disabled (0 = ENABLED, 1=DISABLED)
       OUTA[STATUS_LED_BUS_MUX] := 1

       ' set VGA shared IOs to outputs
       DIRA[23..16]             := %11111111
```

```
      gfx_ntsc.Newline_Term
      gfx_ntsc.Newline_Term
      gfx_ntsc.String_Term( string (''Enter 8-bits values to write''))
      gfx_ntsc.Newline_Term
      gfx_ntsc.String_Term( string (''to port [P23..P16] or -1 to''))
      gfx_ntsc.Newline_Term
      gfx_ntsc.String_Term( string (''exit program.''))
      gfx_ntsc.Newline_Term
      gfx_ntsc.Newline_Term
      gfx_ntsc.String_Term( string (''Hex,Decimal or Binary formats:''))
      gfx_ntsc.Newline_Term
      gfx_ntsc.String_Term( string (''$nn,nnn,%nnnnnnnn respectively.'') )
      gfx_ntsc.Newline_Term

      repeat
        gfx_ntsc.Newline_Term
        gfx_ntsc.String_Term( string (''Write Value?''))

        ' get string from user
        Get_String(@g_sbuffer, 9 )

        ' convert to integer
        g_temp1 := atoi2(@g_sbuffer, 9)

        ' user wants to exit?
        if (g_temp1 == -1)
          quit

        ' write value to port
        OUTA[23..16] := g_temp1

    3:

      ' re-enable VGA buffer(0 = ENABLED, 1=DISABLED)
      OUTA[STATUS_LED_BUS_MUX] := 0

      return ' return to main menu

' end Port_B_Test
```

A fun way to experiment with the port IO demo is to place a single bi-color LED into one of the port A/B headers anywhere. Then set one side of the LED to "1", the other to "0", this will turn the LED on, then switch the data and turn it off. For example, go into the menu and work with Port A (P7..P0) and then place a bi-color LED into port pins [P1, P0] (doesn't matter which way since it will illuminate either way). Then write a %01 to the port, then a %10 to the port; you will see the LED light up red/green (or whatever the colors are).

If on the other hand you want to inject a signal into a port, I suggest connecting a 1 kΩ resistor to 3.3 V on one of the headers with power and then try reading port A or B continuously as you inject the "1" signal into the port. The reason why we use the 1 kΩ resistor, is so that if you short the 3.3 V supply to ground with a straight wire it will reset the C3, but if you put a 1 kΩ resistor inline that will inhibit that problem.

3.3.5 NES Gamepad Demo

The port NES gamepad demo prints out the state of the NES gamepad(s) plugged into the C3 via the Parallax NES Adapter (part # 32368). Basically, the code I used for the pre-programmed Quick-Start test is the same code for the NES demo. So, the first thing is you will need the NES Adapter from Parallax (or make your own) along with at least one NES compatible game pad. Insert the adapter into the C3 as

shown in Figure 1.3 back on page 6. Make sure to line the pins up with the bottom as shown in the figure. The top level file for the demo is named **c3_nesgamepad_demo_010.spin** and you can find it on the FTP site here:

PropC3 \ Sources \ c3_nesgamepad_demo_010.spin

Figure 3.5 below is a screen shot of the start menu and the demo running. It has a little menu that allows you time to insert the adapter and gamepad then press any key to start scanning. As usual, you will need the NTSC monitor and PS/2 keyboard plugged in.

Figure 3.5 — Screen shots of NES Gamepad demo.

(a) Main menu.

(b) Demo running.

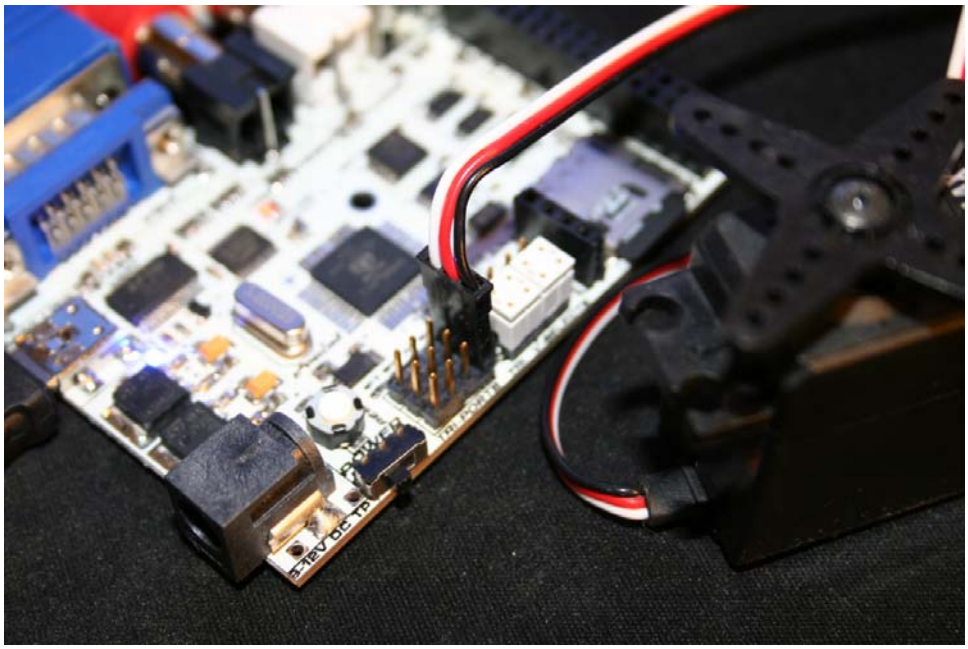
3.3.6 Servo Port Demo

The Propeller, like any other microcontroller, has general IO pins. These pins can drive a digital servo with a standard PWM signal. The problem with this though is that servos require a lot of current, are inductive, and cause current spikes on the power rails when they stop, start, or are stalled. This induces voltage changes to the system power supplies and potentially resetting the processor or wreaking other havoc—all bad things.

Therefore, usually when you want to drive servos you would use a servo driver with a separate power supply and/or buffering. Alas, the C3 doesn't have all that extra hardware, but what it does have is a rather robust servo IO port that has inline signal resistors to dampen the current spikes back to the Propeller chip along with heavy 22 µF capacitors on each servo port. Together these allow you to easily drive up to 4 servos (not simultaneously, without extra bypass capacitors).

As discussed in the Hardware section of the manual, the C3 has 4 servo ports, these are nothing more than 4 headers with 3 pins each that have the signals [P7..P4, power, ground] in that order. If you plug a standard servo into one or more of these ports, for example, the Parallax Standard Servo (#900-00005), then you can control the servo with a standard PWM signal on the control pin (P7..P4). The power for the servo can be sourced by the C3 via the servo port and is selected between [+3.3/5V] by moving the **Tri-Port Voltage Select** jumpers in the top (3.3V) or bottom (5V) positions respectively. With that in mind, for this demo, we are going to take a single Parallax servo and hook it up to servo port P4 (the right most) as shown in Figure 3.6 below.

Figure 3.6 — The hookup of our servo for the demo.



Make sure you connect the servo cable in the proper orientation with **[white, red, black]** top to bottom on the header. These stand for **[signal, power, ground]**.

Then we are going to download the demo program which is named **c3_servo_demo_010.spin** and control the servo position with the local keyboard. The file can be found on the FTP site here:

PropC3 \ Sources \ c3_servo_demo_010.spin

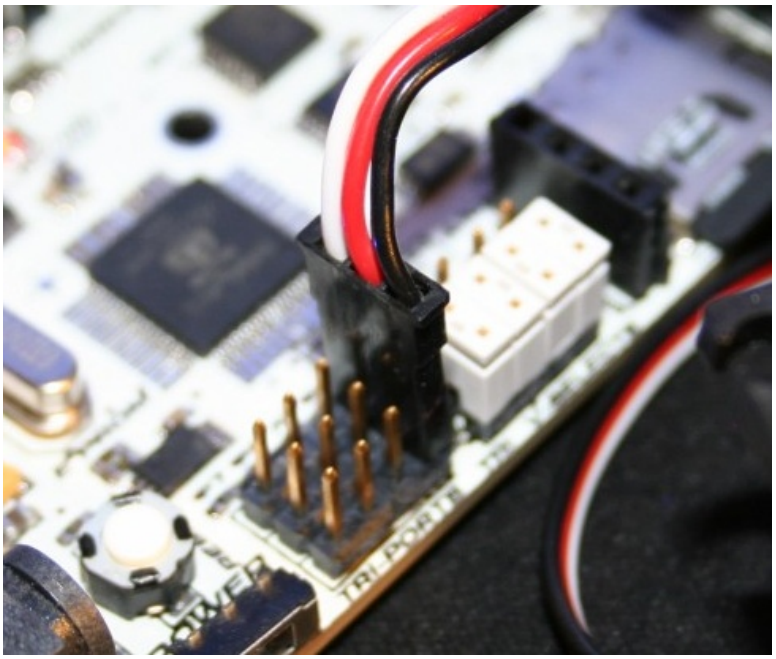
Of course, in addition to a standard servo such as a Parallax (or Futuba), you will need the NTSC monitor and local PS/2 keyboard for text entry. Once you have the program up and running you will see the menu shown in Figure 3.7 below.

Figure 3.7 — Servo demo menu in action.

The program immediately falls into a loop that reads the keyboard and changes the angular position of the servo from 0 to 180 degrees (give or take) based on the right and left arrow keys. Figure 3.8 below

shows an extreme close-up of the servo into the right-most port. Note the orientation and color of the cable wires. Also, note than I have all 4 voltage select jumpers set to 5 V mode (lower position).

Figure 3.8 — Photo of servo cable inserted into servo port P4 on the C3 (extreme close up).



The USB port may not be able to provide enough power to run servo motors, and you may need to provide external power. When connecting external power to the C3, be sure to unplug the USB cable from the C3 before connecting external power.

Controlling servos is beyond the scope of this manual, but the Parallax document for the Standard servo has a little overview and there are many articles, documents, and books on the subject. In fact, the source code of the demo has a very in depth explanation of servo operation. Below is a complete listing of the main demo function in the program:

```
PUB Servo_Demo

  '////////////////////////////////////////////////////////////////////////////
  ' "SERVO" PORT TEST SUITE ////////////////////////////////////////////////////
  '////////////////////////////////////////////////////////////////////////////
{{
   There are 4 servo compatible 3-pin ports on the C3, these are not designed to
   drive servos per se, that is the ports are designed to use the 3-wire "servo"
   connectors that lots of customers like. That said, you "can" drive servos with
   the ports as long as you don't use huge servos with a lot of current draw and
   you don't try to drive multiple servos at the same time. Each servo port has a
   22uF cap across the power feed which is used to store charge when the servo
   needs high currents, this keeps the regulators voltages from dropping too low
   and resetting the processor. Now, if you want to drive servos a lot harder you
   can either get an external driver board, or you can try some of the following
   techniques:

   To drive servos then a good 100-220uf or greater should be put across each power
   feed as well as shunt inductive kick diodes on the servo side as shown below:

          C3 Side             Servo Side            Servo
```

```
                    :
                    :
             50ohm  :                                  _________
   P4..P7 --/\/\/-----------....----------------|         |
                    :                           |    __   |
   +Vcc   ------------------....----------------|   | |   |
            |       :           |     |        |   | |   |
          --- 22uF  :           ---   ----      |   | |   |
          ---       :         C ---    /\  D    |   | |   |
           |        :           |     /__\      |  ( * )  |
           |        :           |      |        |         |
   GND    ------------------....----------------|_________|
                    :
                    :           C = 100-2200uF
                                D = snubber diode, high current schottky or 1N4001 will do

  This little demo simply drives all four servo ports at the same, so you can hook
  up a servo to any port and it will work. You can connect the servo to the servo
  ports P4...P7 with the WHITE lead toward the top of the board (signal) and the BLACK
  lead toward the bottom (ground). Any standard servo will work such as Parallax Standard
  (Futaba servos). The protocol to control a servo is rather simple; you send a square wave
  at a frequency of 1/20ms. The duty cycle of the square wave controls the servo position;
  as you control the HIGH time from 1-2ms the servo will rotate thru its range (180). Here's
  a Parallax reference:

  http://www.parallax.com/Portals/0/Downloads/docs/prod/motors/900-00005StdServo-v2.1.pdf
}}

  gfx_ntsc.Newline_Term
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string ("C3 Servo Demo V1.0") )
  gfx_ntsc.Newline_Term
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string ("Connect servo to any port...") )
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string ("Use LEFT/RIGHT arrow keys to") )
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string ("rotate servo.") )
  gfx_ntsc.Newline_Term
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string ("Hit <ESC> key to exit.") )

  gfx_ntsc.Newline_Term
  gfx_ntsc.Newline_Term
  gfx_ntsc.String_Term( string ("Servo angular position: ") )

  g_tempx := gfx_ntsc.GetX
  g_tempy := gfx_ntsc.GetY

  g_temp1 := 200 ' starting position of servo, not to hyperrotate it
  g_temp2 := 0

  ' update display
  gfx_ntsc.GotoXY( g_tempx, g_tempy )
  gfx_ntsc.Dec_Term( (g_temp1 - 200) / 5  )
  gfx_ntsc.String_Term( string ("  ") )

  ' set servo port IOs to outputs
  DIRA[ SERVO_P7..SERVO_P4 ] := %1111

  ' enter loop
  repeat

    OUTA[ SERVO_P4..SERVO_P7 ] := %1111
```

```
    waitcnt (cnt + (160*g_temp1))                ' the servo signal's high pulse duration

    OUTA[ SERVO_P4..SERVO_P7 ] := %0000
    waitcnt (cnt + 1_600_000 - (160*g_temp1))    ' the servo signal's low pulse duration

    ' user updating position?
    if (kbd.gotkey == TRUE)
      ' get key
      g_key := kbd.key

      ' right or left?
      if (g_key == ASCII_LEFT)

        if (g_temp1 < 1100)
          g_temp1 += 20

      elseif (g_key == ASCII_RIGHT)

        if (g_temp1 > 200)
          g_temp1 -= 20

      elseif (g_key == ASCII_ESC or g_key == ASCII_SPACE )
        ' set servo port IOs to back to inputs
        DIRA[ SERVO_P4..SERVO_P7 ] := %0000

        ' return to caller
        return

      ' update display
      gfx_ntsc.GotoXY( g_tempx, g_tempy )
      gfx_ntsc.Dec_Term( (g_temp1 - 200) / 5  )
      gfx_ntsc.String_Term( string ("  ") )

' end Servo_Test
```

The demo uses Spin alone and crude timing loops to synthesize the duty cycle for the servo, about **20 ms** total cycle time or a frequency 50 Hz with some percentage of that being high (1–2 ms), the rest low. The high period positions the servo angle. This overall signal timing is controlled in the main loop (highlighted in yellow) by a variable named **g_temp1** that is updated by the **<left>** and **<right>** arrow keys. To properly drive a servo, you should use a single cog with virtual timers in a tight loop to generate microsecond accurate timings for as many servos as you wish. But, this little Spin demo gets you up and running if you have never driven a servo before.

3.3.7 SPI Bus API Overview

The SPI bus hardware was discussed in the hardware section, so take a look at that to refresh your memory if you have forgotten the details. To review, the SPI bus consists of a single shared SPI compatible bus composed of 3 primary signals:

- MOSI (SPI_MOSI, P9) – Master Out Slave In.
- MISO (SPI_MISO, P10) – Master In Slave Out.
- CLK (SPI_SCK, P11) – Clock from Master.

Of course the Propeller is master, and there is actually no dedicated SPI hardware per se, this is all bit-banged via the IO ports. In addition to the bus signals that are common to every single SPI device (they are all hooked in parallel), each SPI device needs a chip select line, active low to select it. When it's selected all other devices on the bus must be de-selected and put into a high impedance state.

The selection logic is composed of two IC's; a 74LVC161A 4-bit counter labeled U11 which has 3 of its 4-bit output connected to a 74LVC138A 3-8 bit decoder. Thus, the current count on U11 (0,1,2..7..15) selects one of 8 outputs on the 138 (**Y0n..Y7n**) and asserts it. Thus, we use these signals as the chip selects for channels 0..7 of the SPI system. And each device has a unique SPI channel and physically we connect the SPI select signal to each device. To control the counter there are 2 signals; **count** and **reset**. These signals control the up counting of the counter and the reset back to 0.

That's the hardware interface, so let's talk about the software API to take advantage of the SPI system in a sane manner. But, before we do that, there are some good PDFs in this directory on SPI protocol if you're not proficient in it:

PropC3 \ Docs \ SPI_I2C \ *.*

All we need are a few functions to place the IOs in known state and then a function to select any SPI channel, send and receive SPI data and that's about it. With that in mind, here's a list of the SPI API to get you started on the C3:

SPI API Listing

`PUB SPI_Init` — This function initializes the SPI IO's, counter, and multiplexer, then selects channel 0 and returns.

`PUB SPI_Select_Channel( channel )` — This function sets the active SPI channel chip select on the SPI multiplexer; this is accomplished by first resetting the SPI counter that feeds the SPI select decoder then up counting the requested number of channels.

`PUB SPI_Write_Read( num_bits, data_out, bit_mask)` — This function writes and reads SPI data a bit at a time (SPI is a circular buffer protocol), the data is in MSB to LSB format and up to 32-bits can be transmitted and received, the final result is bit masked by **bit_mask**.

To use the API, first you always call **SPI_Init** at the beginning of your program to put the SPI system in a known state and set the IOs in the correct direction and state. Then, if you want to access a device (SD, FLASH, SRAM, etc.) you select the proper channel with **SPI_Select_Channel (...)** and then finally, write/read to and from the device with the generic function **SPI_Write_Read(...)**.

There is only ONE common shared SPI bus on the C3, so be sure not to have more than one processor (cog) accessing it at the same time.

Now let's take a look at the code for each of the functions (this is an eBook and we have all the space we want!). First, the **SPI_Init** function:

```
PUB SPI_Init
{{
This function initializes the SPI IO's, counter, and mux, selects channel 0 and returns.

PARMS: none.
RETURNS: nothing.
}}
  ' set SPI mux counter IOs up
  DIRA[SPI_SEL_CLK] := 1 ' set to output
  OUTA[SPI_SEL_CLK] := 0

  DIRA[SPI_SEL_CLR] := 1 ' set to output
  OUTA[SPI_SEL_CLR] := 0 ' CLR counter
  OUTA[SPI_SEL_CLR] := 1 ' allow counting
  ' set up SPI lines
```

```
  OUTA[SPI_MOSI]     := 0 ' set to LOW
  OUTA[SPI_SCK]      := 0 ' set to LOW

  DIRA[SPI_MOSI]     := 1 ' set to output
  DIRA[SPI_MISO]     := 0 ' set to input
  DIRA[SPI_SCK]      := 1 ' set to output

  ' set SPI to NULL channel 0
  SPI_Select_Channel( 0 )

' end SPI_Init
```

The function simply sets all the IOs for the SPI bus to the proper state and selects SPI channel 0, the NULL channel, and exits. Next, the **SPI_Select_Channel (...)** function is below:

```
PUB SPI_Select_Channel( channel )
{{
This function sets the active SPI channel chip select on the SPI mux, this is accomplished by
first resetting the SPI counter that feeds the SPI select decoder, then up counting the
requested number of channels.

PARMS:

channel : channel 0 to 7 to enable where the channels are defined as follows
          0 - NULL channel, disables all on/off board devices.
          1 - 32K SRAM Bank 0.
          2 - 32K SRAM Bank 1.
          3 - 1MB FLASH Memory.
          4 - MCP3202 2-Channel 12-bit A/D.
          5 - Micro SD Card.
          6 - Header Interface SPI6.
          7 - Header Interface SPI7.

RETURNS: nothing.

}}

  ' requesting channel 0? If so, easy reset
  if (channel == 0)
    ' clear the 161
    OUTA[SPI_SEL_CLR] := 0 ' CLR counter
    OUTA[SPI_SEL_CLR] := 1 ' allow counting
    return

  ' else non-null channel, count up to channel...

  ' first reset the SPI channel counter
  ' clear the 161
  OUTA[SPI_SEL_CLR] := 0 ' CLR counter
  OUTA[SPI_SEL_CLR] := 1 ' allow counting

  ' now increment to requested channel
  ' clock the 161
   OUTA[SPI_SEL_CLK] := 0

   repeat channel
     OUTA[SPI_SEL_CLK] := 1
     OUTA[SPI_SEL_CLK] := 0

' end SPI_Select_Channel
```

The selection logic begins by testing if the user is requesting channel 0, if so, rather than counting until there is an overflow, a faster method is to reset the counter to 0, which brings me to an optimization. If

we really wanted to, we could have run the counter with a single IO signal as long as we reset it with the C3 reset signal on boot. It would be a little slow, since if we ever wanted to select a channel less that the current channel, we would have to roll all the way around, but with proper code ordering, the impact would be small and we could have saved another IO, but I like the idea of a reset. Anyway, moving on, after the channel 0 test, we simply reset the counter and count up to the proper channel. Now, there is a clear optimization here that we can make and that's to keep a static global that tracks the current channel and if a request is made for the same channel or one that is numerically higher, then all we have to do is clock the counter target—current times.

For example, say the channel is currently set at channel 3 and the user wants channel 7, well, instead of resetting, we can simply clock (7-4) times since we are already at count 4. I kept the function simple without that feature, but add it if you like. The final and most important function is the actual SPI read/write function. As you might know SPI is a circular buffer protocol that writes and reads at the same time, so many times when you read data, you have to write dummy data (usually $00 or $FF) and that confuses newbies to SPI sometimes, they don't see why a read is writing as well? That's why—no matter what when you clock a bit in, you clock a bit out. With that, let's take a look at the final function in the SPI API that performs the read/write:

```
PUB SPI_Write_Read( num_bits, data_out, bit_mask) | data_in
{{
This function writes and reads SPI data a bit at a time (SPI is a circular buffer protocal),
the data is in MSB to LSB format and up to 32-bits can be transmitted and received, the final
result is bit masked by bit_mask

PARMS:

num_bits : number of bits to transmit from data_out
data_out : source of data to transmit
bit_mask : final result of SPI transmission is masked with this to grab the relevant least
significant bits

RETURNS: data retrieved from SPI transmission

}}
  ' clear result
  data_in := 0

  ' now read the bits in/out
  repeat num_bits
    ' drop clock
    OUTA[SPI_SCK] := 0

    ' place next bit on MOSI
    OUTA[SPI_MOSI] := ((data_out >> (num_bits-- - 1)) & $01)

    ' now read next bit from MISO
    data_in := (data_in << 1) + INA[SPI_MISO]

    ' raise clock
    OUTA[SPI_SCK] := 1

  ' set clock and MOSI to LOW on exit
  OUTA[SPI_MOSI]  := 0
  OUTA[SPI_SCK]   := 0

  ' at this point, the data has been written and read, return result
  return ( data_in & bit_mask )

' end SPI_Write_Read
```

The function is well commented, so just peruse it, but I want to bring your attention to a really cool feature. Most SPI transactions are 8 bits, some systems have 9 bits which make interfacing fixed hardware 8-bit SPI transceivers difficult since you must use a pair of 8-bit writes with dummy data to make up the 9-bit write. This brings me to the design feature that I programmed into the function that allows variable-bit-length data to be written. Many SPI transactions are multiplies of a byte; for example, an SPI command might consists of 2 bytes, 3 bytes, or more. Thus, there is no need to have multiple calls to the function when we can process up to 32 bits at once. So, I made the function take as a parameter the number of bits to send and receive. This is really handy when you are programming an SPI device and it says that you need 2 bytes for a command, you can just build up a single 16-bit packet instead of two 8-bit packets! And many commands require the command (1 byte) followed by 1–2 address bytes, so 3 byte commands are now very convenient and even 4 byte commands with 1 byte followed by 3 address bytes. The function handles all of them. For example, say you wanted to write 2 bytes to the SPI bus channel 7 (some external device), here's what you would do:

```
SPI_Select_Channel ( 7 )
SPI_Write_Read( 16, %00000101_00000000, $00 )
```

This code selects channel 7, then it writes the two bytes to the SPI device %00000101_00000000. The bitmask at the end indicates we don't care about any result, just mask it all off to 0's.

That wraps it up for the SPI API, you will see this time and time again in the remaining demos. You will need to copy this code from one of the demos or write something similar if you plan to communicate with the SPI devices on the C3.

3.3.8 SRAM Demo

As mentioned in the Hardware section of the manual, the C3 has two Microchip 32 K byte SPI SRAMs (part #23K256). The SRAMs have a rather robust set of functionality which is outlined in the data sheet. The SRAMs are non-trivial and if you want to expose all their functionality and speed, you should definitely read the data sheet which is located on the FTP site here:

PropC3 \ Docs \ Datasheets \ 23K256.pdf

Of course, to get you started I have created a simple API that allows you to read, write, and fill them with data. With these functions you can access the 64 KB of space and immediately put it to use. However, the driver functions are written in Spin, so if you want to speed them up, you will have to recode to ASM (along with the SPI driver itself). With that said, the SRAM data sheet will give you all the architectural details of the SRAMs themselves, but from our perspective what we are interested in is reading, writing, and understanding the three modes of operation which are:

- Byte Operation — A single byte is read or written.
- Page Operation — A page of 32 bytes is read or written.
- Sequential Operation — Any number of bytes is read or written.

Most SPI devices, whether they be SRAM, FLASH, or other array-based memory devices, have a number of optimized access modes to decrease the amount of SPI traffic to read and write data. For example, to write a single byte to the SRAM, you first need to address the byte (2 bytes) and then write the byte (1 byte) and let's not forget the command itself (1 byte). Therefore, to write a single random access byte is ***four*** bytes of information! That's a huge waste. But, if you need to randomly access the SRAM, that's what to expect. But, what if you want to write a continuous stream of bytes? Maybe you have a 1024 byte buffer in Propeller hub RAM that you want to copy to the SPI SRAM? Well, that's where the sequential mode comes in. In this mode, you write the instruction for sequential access, then the address, then you write byte after byte, so the overhead to write 1024 bytes is nearly 0.

Therefore, a complete API would have these modes and would either allow user selection via function names or parameters or maybe automatic mode selection based on the data sent, size, etc. Some options to think about…

The interesting thing about the Microchip SRAMs is their simplicity in commands. They only have 4 commands as shown in Table 3.1 below.

Table 3.1 — SRAM commands.

Instruction Name	Value	Description
READ	0000 0011b	Read data from memory array beginning at selected address.
WRITE	0000 0011b	Write data to memory array beginning at selected address.
RDSR	0000 0101b	Read STATUS register.
WRSR	0000 0001b	Write STATUS register.

You may note there is no mention of the 3 modes of operation; byte, page, sequential. These are set once by writing the status register and then the chip is in that mode until you change it—read the data sheet to learn more.

SRAM API Overview

Our little API is very simple and consists of a few functions, here are their prototypes:

`PUB SRAM_Init ( ... )` — Initializes the SRAMs and sets them to sequential mode.

`PUB SRAM_Write( ... )` — Writes bytes of data from a buffer to either SRAM.

`PUB SRAM_Fill( ... )` — Fills either SRAM with a constant value.

`PUB SRAM_Read( ... )` — Reads data from either SRAM into a buffer.

`PUB SRAM64K_MREAD( ... )` — Models the pair of 32 KB SRAMs as a single memory of 64 KB and reads a byte from it.

`PUB SRAM64K_MWRITE( ... )` — Models the pair of 32 KB SRAMs as a single memory of 64 KB and writes a single byte to it.

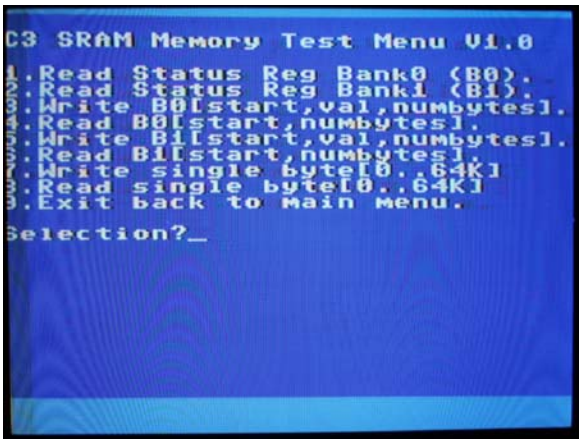
So, to use the SRAMs you first make a call to **SRAM_Init**, and then you make calls to **SRAM_Write**, SRAM_Read, etc. as needed. Also, since there are two SRAMs, you call the first write, read, and fill functions with a parameter that indicates the SRAM bank:

```
  SPI_CHAN_SRAM0         = 1 ' 32K SRAM Bank 0.
  SPI_CHAN_SRAM1         = 2 ' 32K SRAM Bank 1.
```

But, with the later functions **SRAM64K_MREAD** and **SRAM64K_MWRITE**, these functions take an address from 0..64 KB and then auto-select the correct SRAM bank for the internal SPI calls. Therefore, you can use this pair of functions almost as if they are accessing a 64 K byte array which is a nice abstraction to make.
With that in mind, let's take a look at the demo. Its name is **c3_sram_demo_010.spin** and you can find the file in the FTP site here:

PropC3 \ Sources \ c3_sram_demo_010.spin

Figure 3.9 — SRAM Demo Menu



This demo has the most complex menu thus far as shown in Figure 3.9. Let's take a quick look at each function and how it works one menu item at a time:

1. Read Status Reg Bank0 (B0) — This simply prints out the status register for SRAM bank 0.

2. Read Status Reg Bank1 (B1) — This simply prints out the status register for SRAM bank 1.

3. Write B0[start, val, numbytes] — This writes SRAM bank 0 with the value **val** at the starting location **start** with **numbytes** bytes.

4. Read B0[start, numbytes] — This reads **numbytes** bytes from SRAM bank 0 starting at memory location **start**.

5. Write B1[start, val, numbytes] — This writes SRAM bank 1 with the value **val** at the starting location **start** with **numbytes** bytes.

6. Read B1[start, numbytes] — This reads **numbytes** bytes from SRAM bank 1 starting at memory location **start**.

7. Write single byte[0..64K] — This function uses the 64K functions to write a single byte to any location from 0 to 64K.

8. Read single byte[0..64K] — This function uses the 64K functions to read a single byte from any location 0 to 64K.

9. Exit back to main menu — Just loops back to the menu.

Now, a couple tips about using the menu. Most items have prompts, you can edit your inputs, that means backspace, and remember you are allowed to use hex $xx, binary %xxxxxxxx or decimal input values. For example, when you write the SRAM, you might want to write 100 bytes with the value $FF starting at $100. These are all valid inputs and the parser will understand.

SRAM API Code Listing

Now, let's take a look at the SRAM functions themselves (this is an eBook, so plenty of room).

```
PUB SRAM_Init ( mode ) | _spi_word
{{
This function initializes both banks of the Microchip 23K256 SRAMs
serial SPI SRAMs'. The initialization simply sets the STATUS register
of each SRAM into "byte", "page" or "sequential" read/write mode to allow more
efficient access of the SRAMs. The SRAM API currently relies on "sequential"
mode, so if yoo want to use them, this function should be called with SPI_MODE_SEQUENTIAL

The 23K256 supports a number of commands, we have
only implemented a couple here in these API functions (single byte read/write,
multiple byte read/write). However, there are many others you can implement, find
out more by reviewing the chip specs and data sheets found here:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en539039
http://ww1.microchip.com/downloads/en/DeviceDoc/22100E.pdf

PARMS:  mode - %00_00000_1 = SPI_MODE_BYTE
               %10_00000_1 = SPI_MODE_PAGE
               %01_00000_1 = SPI_MODE_SEQUENTIAL

RETURNS: Nothing.

}}

  ' first set SRAM bank 0 to sequential access mode (see page 6 of data sheet)
  SPI_Select_Channel( SPI_CHAN_SRAM0 )

  ' write status register command "sequential access mode" along with disable "HOLD"
  ' to setup SRAM for sequential reading/writing
  SPI_Write_Read( 16, %0000000_1 << 8 |  mode , $FF )

  ' now select SRAM bank 1
  SPI_Select_Channel( SPI_CHAN_SRAM1 )

  ' write status register command "sequential access mode" along with disable "HOLD"
  ' to setup SRAM for sequential reading/writing
  SPI_Write_Read( 16, %0000000_1 << 8 |  mode , $FF )

  ' and finally de-select all SPI devices
  SPI_Select_Channel( 0 )

' end SRAM_Init

' ////////////////////////////////////////////////////////////////////////////

PUB SRAM_Write( bank, addr, num_bytes, src_buffer_ptr) | _index, _data, _spi_word
{{
This function writes a number of bytes to either bank of the Microchip 23K256
serial SPI SRAMs' from a buffer. The 23K256 supports a number of commands, we have
only implemented a couple here in these API functions (single byte read/write,
multiple byte read/write). However, there are many others you can implement, find
out more by reviewing the chip specs and data sheets found here:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en539039
http://ww1.microchip.com/downloads/en/DeviceDoc/22100E.pdf


PARMS:  bank           - [0,1] the SRAM bank to write to, there are two banks, each holds 32K
                       bytes
```

```
        addr           - address in SRAM to write to [0...32767]
        num_bytes      - number of bytes to write from source buffer to SRAM
        src_buffer_ptr - pointer to the buffer of bytes to write to SRAM

RETURNS: number of bytes written

}}

  ' test if there is anything to write?
  if (num_bytes => 1)

    ' select SPI channel for requested SRAM bank
    SPI_Select_Channel( SPI_CHAN_SRAM0 + bank )

    ' set instruction word ( write sequential byte:8 | address:16 )
    _spi_word := (%00000010 << 16) + (addr)

    ' and now send command to SRAM for sequential writing
    SPI_Write_Read( 24, _spi_word, $FF )

    ' write the bytes into the SRAM
    repeat _index from 0 to num_bytes-1

      ' get byte to write
      _data := byte[ src_buffer_ptr][ _index ]

      ' write the byte
      SPI_Write_Read( 8, _data, $FF )

    ' de-select SRAM SPI channel
    SPI_Select_Channel( 0 )

    ' return bytes written
    return( num_bytes )

  else
    ' catch error
    return 0

' end SRAM_Write

' ////////////////////////////////////////////////////////////////////////////

PUB SRAM_Fill( bank, addr, num_bytes, value) | _index, _data, _spi_word
{{
This function writes a constant value to either bank of the Microchip 23K256
serial SPI SRAMs' from a buffer. The 23K256 supports a number of commands, we have
only implemented a couple here in these API functions (single byte read/write,
multiple byte read/write). However, there are many others you can implement, find
out more by reviewing the chip specs and data sheets found here:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en539039
http://ww1.microchip.com/downloads/en/DeviceDoc/22100E.pdf


PARMS:  bank           - [0,1] the SRAM bank to write to, there are two banks, each holds 32K
                       bytes
        addr           - address in SRAM to write to [0...32767]
        num_bytes      - number of bytes to write from source buffer to SRAM
        value          - byte value to write to SRAM

RETURNS: number of bytes written

}}
```

```
  ' test if there is anything to fill?
  if (num_bytes => 1)

    ' select SPI channel for requested SRAM bank
    SPI_Select_Channel( SPI_CHAN_SRAM0 + bank )

    ' set instruction word ( write byte sequential:8 | address:16 )
    _spi_word := (%00000010 << 16) + (addr)

    ' setup for sequential byte write, this includes the command and address only
    SPI_Write_Read( 24, _spi_word, $FF )

    ' now the chip is ready to receive bytes (as many as we want to send)
    repeat _index from 0 to num_bytes-1
      SPI_Write_Read( 8, value, $FF )

    ' de-select SRAM channel
    SPI_Select_Channel( 0 )

    ' return bytes written
    return( num_bytes )

  else
    ' catch error
    return 0

' end SRAM_Fill

' ////////////////////////////////////////////////////////////////////////////

PUB SRAM_Read(bank, addr, num_bytes, dest_buffer_ptr) | _index, _data, _spi_word
{{

This function reads a number of bytes from either bank of the SRAM into a buffer.

The 23K256 supports a number of commands, we have only implemented a couple here
in these API functions (single byte read/write, multiple byte read/write).
However, there are many others you can implement, find out more by reviewing
the chip specs and data sheets found here:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en539039
http://ww1.microchip.com/downloads/en/DeviceDoc/22100E.pdf

PARMS:  bank            - [0,1] the SRAM bank to read from, there are two banks, each holds
                          32K bytes
        addr            - address in SRAM to read from [0..32767]
        num_bytes       - number of bytes to read from SRAM into destination buffer
        dest_buffer_ptr - pointer to the buffer of bytes to read SRAM into

RETURNS: number of bytes read

}}

  ' test for anything to read?
  if (num_bytes => 1)

    ' select SPI channel for requested SRAM bank
    SPI_Select_Channel( SPI_CHAN_SRAM0 + bank )

    ' set instruction word ( read byte sequential:8 | address:16 )
    _spi_word := (%00000011 << 16) + (addr)

    ' setup for sequential byte read, this includes the command and address only
```

```
    SPI_Write_Read( 24, _spi_word, $FF )

    ' now the chip is ready to be read from
    repeat _index from 0 to num_bytes-1

      ' read next byte from SRAM into receiving buffer
      byte [dest_buffer_ptr][_index] := SPI_Write_Read( 8, $00, $FF )

    ' de-select SRAM channel
    SPI_Select_Channel( 0 )

    ' return bytes read
    return( num_bytes )
  else
    ' catch error
    return 0

' end SRAM_Read

' /////////////////////////////////////////////////////////////////////////////

PUB SRAM64K_MREAD( addr ) | _spi_word, _data
{{
This function reads a single byte from either bank of the Microchip 23K256
serial SPI SRAMs'. This function is different from the SRAM_Read function
in that it accesses the two banks of SRAM as a contiguous region of 64K bytes
for you, and thus can be though of as a crude "array" access macro, rather
than a function. Ultimately, this should be converted to ASM along with
all the other important peripheral access methods.

The 23K256 supports a number of commands, we have
only implemented a couple here in these API functions (single byte read/write,
multiple byte read/write). However, there are many others you can implement, find
out more by reviewing the chip specs and data sheets found here:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en539039
http://ww1.microchip.com/downloads/en/DeviceDoc/22100E.pdf


PARMS:  addr - address in SRAMs to write to [0...65535].

RETURNS: The 8-bit data from the SRAM.

}}
    ' the only tricky thing here is we need to take an address 0..65535 and then
    ' decide which bank it refers to and which byte in that bank, basically, some
    ' bit masking and shifts to accomplish this

    ' select SPI channel for requested SRAM bank
    SPI_Select_Channel( SPI_CHAN_SRAM0 + (addr >> 15) )

    ' set instruction word ( read byte sequential:8 | address:16 | dummy:8 )
    _spi_word := (%00000011 << 24) + (addr << 8) + 0

    ' read the byte of data
    _data := SPI_Write_Read( 32, _spi_word, $FF )

    ' de-select SRAM SPI channel
    SPI_Select_Channel( 0 )

    ' return data
    return ( _data )

' end SRAM64K_READ
```

```
' /////////////////////////////////////////////////////////////////////////////

PUB SRAM64K_MWRITE( addr, data ) | _spi_word
{{
This function writes a single byte to either bank of the Microchip 23K256
serial SPI SRAMs'. This function is different from the SRAM_Write|Fill functions
in that it accesses the two banks of SRAM as a contiguous region of 64K bytes
for you, and thus can be though of as a crude "array" access macro, rather
than a function. Ultimately, this should be converted to ASM along with
all the other important peripheral access methods.

The 23K256 supports a number of commands, we have
only implemented a couple here in these API functions (single byte read/write,
multiple byte read/write). However, there are many others you can implement, find
out more by reviewing the chip specs and data sheets found here:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en539039
http://ww1.microchip.com/downloads/en/DeviceDoc/22100E.pdf


PARMS:  addr - address in SRAMs to write to [0...65535].
        data - the byte to write.

RETURNS: Nothing.

}}

    ' the only tricky thing here is we need to take an address 0..65535 and then
    ' decide which bank it refers to and which byte in that bank, basically, some
    ' bit masking and shifts to accomplish this

    ' select SPI channel for requested SRAM bank
    SPI_Select_Channel( SPI_CHAN_SRAM0 + (addr >> 15) )

    ' set instruction word ( write sequential byte:8 | address:16 | data: 8)
    _spi_word := (%00000010 << 24) + ( (addr & $7FFF) << 8) + data

    ' write the byte of data
    SPI_Write_Read( 32, _spi_word, $FF )

    ' de-select SRAM SPI channel
    SPI_Select_Channel( 0 )

' end SRAM64K_WRITE
```

The functions are very short, only a few lines each (mostly comments), but you will notice a pattern in them; they all select the SPI channel, then build the command in the binary format:

[word+address+[data]]

Then start the write/read process. Also, remember addresses need to be from [0..32K], data is always 8-bit bytes, and the bank is 0/1. Of course, the 64K functions take addresses [0..64K] and no bank. First, to use the SRAM functions, you need to have all the SPI bus functions in your source and make the call to **SPI_Init**, then you need to make a call to **SRAM_Init (...)** with the proper parameter as shown below:

```
  always do this once at the top of your code if you want to use the SPI bus,
  also make sure to include the SPI bus functions source to read/write data..
SPI_Init

  initialize the SRAMs and set them to sequential mode, the API requires this
SRAM_Init (SPI_MODE_SEQUENTIAL)
```

The SRAM functions obviously can be optimized quite a bit algorithmically and by using ASM as well. But, for now, start with the Spin routines, they should be fast enough to experiment around then you can upgrade to ASM or whatever you like to do.

3.3.9 Simple A/D Demo

Analog to digital conversion is becoming more and more desired in control applications. Even low cost microcontrollers come equipped with one or more analog to digital convertors that operate asynchronously to the processing core. However, the philosophy of the Propeller chip is **virtualization** of hardware peripherals through software emulation; therefore, the Propeller does not have any A/D (or D/A for that matter). Our options are to emulate analog to digital conversion with software ***or*** use an external A/D.

Using the counter modules with simple R/C timing circuits is a low cost viable solution, however it requires one cog and 4 I/O pins for two channels. Wasting I/O pins isn't something we want to do with the C3. Rather, the C3 has an external, very accurate 12-bit, dual channel successive approximation SPI A/D based on the Microchip MCP3202 chip. The MCP3202 is a very capable chip from an operational point of view. It has two analog inputs labeled channel 0 and 1. These can sample a single ended signal or be coupled into a differential mode of operation. Also, since we are running the A/D at 5V, the input signals can range from 0–5 V and the sampling occurs at a rate of 100 ksps (kilo samples per second). Thus, you can easily sample audio signals in real-time as well as make a poor man's two channel logic analyzer or o-scope with the C3!

Analog to digital conversion is a very complex subject, so I suggest you read up on the nuances of it if you haven't worked with it before. That said, the data sheet is a must if you want to program the MCP3202 yourself, so please review it. You can find the data sheet located on the FTP site here:

PropC3 \ Docs \ Datasheets \ 23K256.pdf

As usual, I have created a simple API to read the MCP3202 A/D channels in single-ended mode. The sample API is used as part of the demo program which is named **c3_analog_digital_demo_010.spin** and is located on the FTP site here:

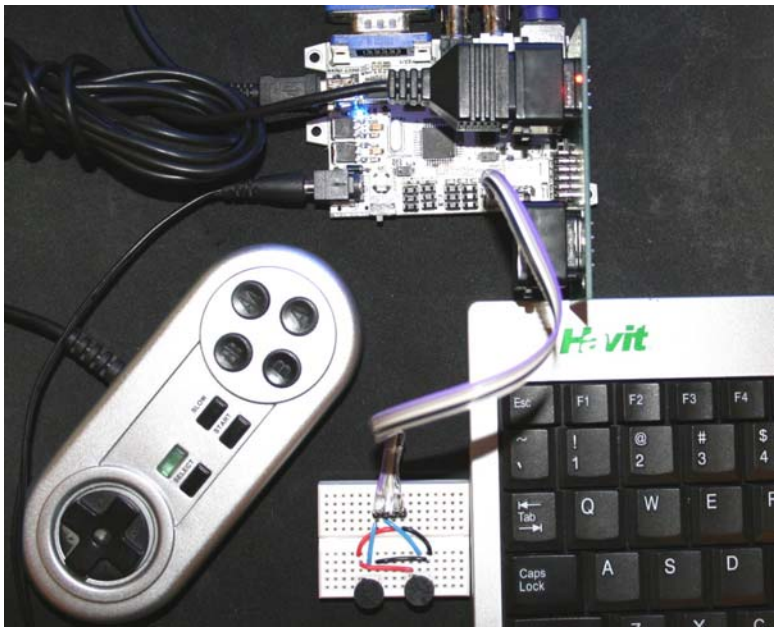
PropC3 \ Sources \ c3_analog_digital_demo_010.spin

Figure 3.10 — Simple A/D demo in action.

Figure 3.10 shows a screen shot of the demo running. I tried to use a little ASCII art to represent the actual signal header on the C3, so there is no confusion about what goes where. As you can see, you need to inject the signal(s) into pin(s) 3, 4, both or one. If you leave one open, I suggest grounding it so

it doesn't pick up noise. A simple circuit you can make is to use a pair of potentiometers (1 kΩ to100 kΩ) and apply the ground and +5 V to the outside contacts (pins 1,3) and then the wiper (pin 2) in the middle will output 0..5 V as you rotate it back and forth. Figure 3.11 shows a test rig setup that I constructed to test the A/D on the C3.

Figure 3.11 — A/D test rig.



Referring to Figure 3.11, I built the circuit on a mini-solderless breadboard (Parallax Part #700-00012). This is very convenient for quick and dirty designs. Also, I made a little colored cable (purple, gray, white, black) with a header on it to connect to the analog to digital port. Once you have your C3 setup this way or similar, then you can let the program run as shown in Figure 3.10(b), it simply prints the channel values out 0..4095.

Input Impedance — Although I said that you can use any value of a potentiometer to generate the analog voltage from 0 to 5 V, there is a whole section on input impedance in the data sheet for the MCP3202 which outlines best practices for impedance matching your inputs. In general, the lower your input's impedance is the better. But, refer to the MCP3202 datasheet, especially Figure 4.1, page 12.

Analog to Digital API Listing

The MCP3202 is accessed via the SPI bus channel 4. Reading the A/D is simply a matter of sending the correct SPI request and then reading the data back. Referring to Table 5.1 in the MCP3202 datasheet, reading the value of either channel requires a single SPI transaction! You simply send a few control bits that indicate single or differential mode and which channel, and the device returns a 12-bit value [0..4095] which represents the voltage on the channel relative to the Vdd/Vref pin. The C3 uses 5 V for Vdd/Vref, thus 0 V on a channel results in 0 being returned, and 5 V on the channel input results in 4095 being returned. The call is so simple a function isn't even required, a single line of code reads the A/D with a generic SPI call:

```
SPI_Write_Read( 17, %0001_1C_1_0000000000000, $FFF )

' (where C=(0|1) refers to channel 0,1)
```

Nevertheless, there are two functions I provided to access the A/D converter:

PUB AD_Init — Initializes the A/D converter.

PUB AD_Read(...) — Reads channel 0/1 in single ended mode 5 V range.

Here's the complete source code for the functions:

```
PUB AD_Init

{{
This function initializes the MCP3202 A/D converter. Currently, does nothing
more than asserts the CS then de-asserts it.

PARMS:  None.

RETURNS: Nothing.

}}

  ' select and de-select the A/D device on the SPI bus just to wake it up and
  ' make sure we know the state of the A/D
  SPI_Select_Channel( 0 )
  SPI_Select_Channel( SPI_CHAN_AD )
  SPI_Select_Channel( 0 )

' end AD_Init

' ////////////////////////////////////////////////////////////////////////////

PUB AD_Read( channel ) | _data

{{
This function reads from the MCP3202 the sent AD channel (0,1) in single ended mode and
returns a 12-bit integer [0..4095] representing the value.

PARMS:  channel - [0,1] which channel to read.

RETURNS: returns value of A/D converter [0..4095]

}}

  ' read channel 0, SPI transaction is 17 bits total,
  ' control bits indicate single ended mode, channel (0,1)
  SPI_Select_Channel( SPI_CHAN_AD )
  _data := SPI_Write_Read( 17, %0001_10_1_0000000000000 | channel << 14, $FFF )
  SPI_Select_Channel( 0 )

  ' return A/D value
  return ( _data )

' end AD_Read
```

As usual, to use any SPI-related functions make sure to include the SPI bus API library sources and make a call to **SPI_Init(...)** as your first call before making any other calls.

3.3.10 A/D Plus SRAM Demo

This demo is really nothing more than a hybrid of the SRAM demo and the A/D demo, so look to those for details about the SRAM and A/D converter. This hybrid demo illustrates crude data logging and records the values A/D converter 0 as a function of time. As its recording, it writes the 12-bit data into

the SRAM bank 0,1 in the format [8:4]-bits. Then the demo stops recording and plays back the data to the screen. The idea here is the analog input could have been audio or something else, and instead of printing numbers to the screen, it could control a servo, or speaker, or whatever.

The name of the demo is **c3_ad_sram_demo_010.spin** and you can find it on the FTP site here:

PropC3 \ Sources \ c3_ad_sram_demo_010.spin

To run the demo you will need the NTSC monitor and PS/2 keyboard as usual, along with some source of analog input on Channel 0 of the Analog to Digital port. Figure 3.12 shows the demo start up, recording, and playback respectively left to right.

Figure 3.12 — The Hybrid A/D plus SRAM Demo

(a) Start up. (b) Recording phase. (c) Playback phase.

The demo records 2000 samples at a rate of 10–20Hz (it's only slow because it has to print to the screen), and then plays back at the same rate, so you can see the data as it scrolls by. Of course, we have enough memory to record 32,768 samples at 12 bits without compression. But, right now the demo writes a byte to bank 0 and the remaining 4 bits of the sample to bank 1 of the SRAM. The remaining 4 bits of each byte in bank 1 is unused.

With simple compression such as wavelet, run length, etc. you can get 2–10x compression ratios easily. Therefore the 64 KB of SRAM at 8 bits per sample could potentially store a minute or more of mono audio data at 11kHz sampling rates.

3.3.11 FLASH Memory Demo

The FLASH memory on the C3 consists of a single 8 mega-bit SPI FLASH which is organized at 4096 pages of 256 bytes. The chip we are using is the Atmel **AT26DF081A-SSU**, for more information on the hardware design review the Hardware manual and make sure to read the data sheet located here on the FTP site:

PropC3 \ Docs \ Datasheets \ AT26DF081A.pdf

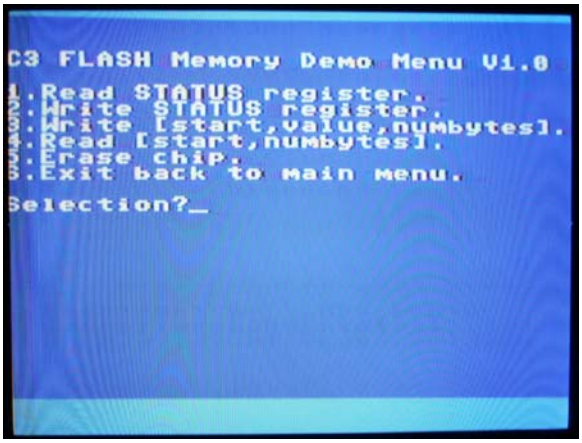
The FLASH memory is the most complex chip on the C3 (other than the Propeller chip itself). In fact, the FLASH chip has a small state machine inside which manages all the reads, writes, and power states. More or less, the FLASH is a poor mans solid state disk drive. The AT26DF081 has around 20 commands, advanced hardware and software data protection, single or sequential byte access modes and the ability to withstand 100,000+ programming cycles before the chip starts to degrade. In fact, after 100,000 cycles the only adjustment you have to do when writing is potentially do a read-back and re-write operation. I personally have found that you can get a million cycles out of these chips no problem. Of course, you can read them forever without any degradation.

Understanding and using FLASH memories is very important since the introduction of FLASH and solid state drives into PC computing. The issue of finite write cycles is addressed in many operating systems with wear leveling which is a technique where data is written to different parts of a FLASH drive that is erased frequently to increase the longevity of the FLASH medium. It is very common on embedded systems.

The API for the FLASH is rather complex, but I have insulated you from as much pain as possible. We will take a look at the API, but first let's see the demo program in action. The name of the demo is **c3_flash_demo_010.spin** and it's located on the FTP site here:

PropC3 \ Sources \ c3_flash_demo_010.spin

Figure 3.13 — The FLASH demo program in action



Referring to Figure 3.13, the demo starts off with a menu consisting of 5 primary options. These options have functions that exercise the API, so you can use them for reference in your code to get you started. Let's take an in depth look at them, so it's clear how to use the demo.

1. Read STATUS register — This option reads the internal STATUS register and prints it to the screen. The STATUS register controls global protection as well as indicates status of any pending operations. You can read more about it on page 19 of the Atmel AT26DF081A data sheet mentioned earlier.

2. Write STATUS register — This option allows you to directly write to the STATUS register and effect any changes you might like to make to the chip. For example, one of the most important things to understand about FLASH memories is they go through a lot of trouble to protect themselves from accidental writes. Therefore, before any write operation you have to set a "write enable" bit in the STATUS register then perform the write operation itself.

3. Write [start,value, numbytes] — This option allows you to write any value to a number of bytes at any starting location. The option as usual will guide you through the inputs and takes either decimal (default), hex inputs ($xx..) or binary (%xx..) for your convenience. Note: you can only write a memory location that has not been written to. Once a memory location has been written to, you can't alter it, you must erase the block, sector, or the entire chip first to clear the memory location.

4. Read [start,numbytes]	This option allows you to read any portion of the FLASH memory and print it to the screen in a nice hex table format. You send the starting address and number of bytes you wish to read.
5. Erase chip	This option performs a bulk erase of the FLASH chip and resets all memory cells back to $FF. It usually takes a few seconds.
6. Exit back to main menu	Just loops back to this menu.

FLASH API Listing

Before we get into the API, here are a couple more interesting pieces of information. The FLASH chip runs up to 70 MHz on the SPI bus and is compatible with the JEDEC spec for FLASH memory and pin outs. In other words, if you have written a FLASH driver for say a Microchip, or ST part, chances are it will take moments to port it to this chip. That said, FLASH chips have complex functionality and writing a generic driver that supports all the functions would be a waste. Therefore, I have provided a starter API here to get you going, so you can write, read, erase the chip, and start experimenting. I am sure there are numerous serial FLASH drivers you can find for the chip and even some in Spin! Here's a list of the API functions:

`PUB Flash_Read( flash_start_addr, buffer_ptr, num_bytes)` — Reads bytes from flash into a buffer.

`PUB Flash_Write( flash_start_addr, buffer_ptr, num_bytes)` — Writes bytes to flash from buffer.

`PUB Flash_Erase( block )` — Erases a block of the flash.

`PUB Flash_Erase_Chip( wait )` — Erases the entire chip.

`PUB Flash_Close` — Closes the flash (not much now, but this is a good place to de-allocate things in the future).

`PUB Flash_Open` — Opens the flash (again not much now, but this is a good place to allocate things in the future).

`PUB Flash_Write_Status ( status )` — Writes the STATUS register of the flash memory.

`PUB Flash_Read_Status` — Reads the STATUS register of the flash memory.

These functions more or less give you access to the FLASH memory and are based on a handful of the FLASH commands (read the datasheet to learn more).

Additionally, this demo has a number of definitions/constants in the **CON** to help deal with the chip, so pay attention to those. If you want to pull the source from this file to build an application then you will need those constants (copied below for reference):

```
'/////////////////////////////////////////////////////////////////////////////
' FLASH MEMORY DEFINES ////////////////////////////////////////////////////////
'/////////////////////////////////////////////////////////////////////////////

' defines for the proper FLASH installed on C3 4MB, 8MB, 16MB
' 4Mb (512K Bytes)
AT26F004        = 4   ' numeric codes for flash version
AT25DF041A      = 4

' 8Mb (1M Bytes)
```

```
AT26DF081A      = 8
AT25DF081A      = 8

' 16 Mb (2M Bytes)
AT25DF161       = 16

' set type of FLASH ROM currently installed in C3 (8MBit) for conditional
'compilation of code -- not supported in SPIN, so manually done below :(
FLASH_ROM             = AT26DF081A
NUM_FLASH_ROM_SECTORS = 19 ' 11 for 4MB, 19 for 8MB, must be set manually due to lack of
                           ' conditional compilation


' Atmel 2F004/8xxx series basic commands
WRITE_ENABLE     = $06
WRITE_DISABLE    = $04
READ_STATUS      = $05
WRITE_STATUS     = $01
READ_DATA        = $03

SEQ_PROGRAM      = $AF
BYTE_PROGRAM     = $02

BLOCK_ERASE4     = $20
BLOCK_ERASE32    = $52
BLOCK_ERASE64    = $D8

SECTOR_ERASE     = $20

CHIP_ERASE       = $60
JEDEC_ID         = $9F

PROTECT_SECTOR   = $36
UNPROTECT_SECTOR = $39

' custom commands we make up, not part of chip spec, just used here, start at code 0xF0
UNPROTECT_CHIP   = $F0
```

Additionally, there is a data structure that holds starting addresses of sectors. The Atmel chip like many others has a number of quirks, one of them is that instead of having same sized sectors, it has different sized sectors at different addresses, so we need a list of these since the code to generate them procedurally would be 10x longer than a simple table. Thus, you need the data structure "sector_addresses" in the **DAT** section below:

```
    ' 19 sectors in 8MB FLASH
    sector_addresses long $00000   ' 64K sector 0
                     long $10000   ' 64K sector 1
                     long $20000   ' 64K sector 2
                     long $30000   ' 64K sector 3
                     long $40000   ' 64K sector 4
                     long $50000   ' 64K sector 5
                     long $60000   ' 64K sector 6
                     long $70000   ' 64K sector 7
                     long $80000   ' 64K sector 8
                     long $90000   ' 64K sector 9
                     long $A0000   ' 64K sector 10
                     long $B0000   ' 64K sector 11
                     long $C0000   ' 64K sector 12
                     long $D0000   ' 64K sector 13
                     long $E0000   ' 64K sector 14

                     long $F0000   ' 16K sector 15
                     long $F4000   '  8K sector 16
```

```
                    long $F6000   '  8K sector 17

                    long $F8000   ' 32K sector 18
```

As usual, we have infinite paper since this is an eBook, so without further ado, below is the complete FLASH API (note you will notice its very C/C++ looking, this is because I ported this library from another C/C++ based design I developed for another processor).

```
' /////////////////////////////////////////////////////////////////////////////
' FLASH MEMORY API (ported from my C/C++ libraries note syntax leftovers!)
' /////////////////////////////////////////////////////////////////////////////

PUB Flash_Read_Status | _spi_data
{{
Reads the FLASH memory status register

PARMS: None.
RETURNS: status register value
}}

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, READ_STATUS, $FF )

  ' now read data returned
  _spi_data := SPI_Write_Read( 8, $FF, $FF )

  '// disable SPI interface
  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )

  ' return results
  return ( _spi_data )

' // end Flash_Read_Status

' /////////////////////////////////////////////////////////////////////////////

PUB Flash_Write_Status ( status ) | _spi_data
{{
Writes the FLASH memory status register

PARMS:

status - value to update the status register with

RETURNS: None.
}}

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, WRITE_STATUS, $FF )
  _spi_data := SPI_Write_Read( 8, status, $FF )

  '// disable SPI interface
  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )

  ' return success for now
  return ( 1 )

' // end Flash_Write_Status
```

```
' /////////////////////////////////////////////////////////////////////////////

PUB Flash_Open | _index, _spi_data, _flash_start_addr
{{

This functions "opens" the FLASH memory simply by unprotecting all sectors starting at
address 0

PARMS: None.
RETURNS: None.

}}


  ' use global unprotect to unprotect entire chip at once, write "0" to bits 2,3,4,5
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, WRITE_ENABLE, $FF )
  SPI_Select_Channel( 0 )

  Flash_Write_Status ( Flash_Read_Status & %11_0000_11 )
  Delay_MS( 10 )
  ' // return success always
  return( 1 )

{


  ' this code is used to iterate thru all sectors manually and have more control
  ' iterate thru all sectors and unprotect them
  repeat _index from 0 to NUM_FLASH_ROM_SECTORS-1

    '// STEP 1: enable writing to chip  ////////////////////////////////////////
    '// enable SPI interface

    '// set CS to SPI select channel 3 (FLASH)
    SPI_Select_Channel( SPI_CHAN_FLASH )
    _spi_data := SPI_Write_Read( 8, WRITE_ENABLE, $FF )

    '// disable SPI interface
    '// set CS to SPI select channel 0 (null)
    SPI_Select_Channel( 0 )

    Delay_MS( 10 )

    '// unprotect sector  ///////////////////////////////////

    '// enable SPI interface

    '// set CS to SPI select channel 3 (FLASH)
    SPI_Select_Channel( SPI_CHAN_FLASH )
    _spi_data := SPI_Write_Read( 8, UNPROTECT_SECTOR, $FF )

    '// get sector address
    _flash_start_addr := sector_addresses[ _index ]

    '// write sector address, there are only a handful of sectors since each is 4-64K
    _spi_data := SPI_Write_Read( 8, _flash_start_addr >> 16, $FF )
    _spi_data := SPI_Write_Read( 8, _flash_start_addr >> 8, $FF )
    _spi_data := SPI_Write_Read( 8, _flash_start_addr >> 0, $FF )

    '// disable SPI interface
    '// set CS to SPI select channel 0 (null)
    SPI_Select_Channel( 0 )
```

```
    ' delay 10
    Delay_MS( 10 )

    ' // end for index

  ' // return success always
  return( 1 )
 }

' // end Flash_Open

' ////////////////////////////////////////////////////////////////////////////

PUB Flash_Close | _index, _spi_data, _flash_start_addr
{{
This function "closes" the FLASH by protecting all of the sectors on the flash chip.

PARMS: None.
RETURNS: None.

}}

  '// protecting the sectors seems to have a side effect, we are not considering???
  '// we need to read the data sheet more, for now, just return, and don't protect
  '// all the sectors, this is overkill anyway for now ...
  return(1)


' // end Flash_Close

' ////////////////////////////////////////////////////////////////////////////

PUB Flash_Erase_Chip( wait ) | _spi_data
{{
This function erases the entire chip. The erase command takes a considerable amount
of time, thus this function issues the command and either returns immediately or waits.
To determine if the erase is complete the function polls the status register bit 0
which is the RDY/BSY bit, 0=device is idle, 1=device is busy

PARMS: wait - boolean, 1=wait for erasure to complete, 0-return immediately.

RETURNS: None.

}}
  '// enable SPI interface

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, WRITE_ENABLE, $FF )

  '// disable SPI interface

  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, CHIP_ERASE, $FF )

  '// disable SPI interface

  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )
```

```
  ' test if user requested wait for completion?
  if (wait)
    repeat
      ' erase operation still ongoing?
     if ((Flash_Read_Status & $01)==0)
       return (1)

  '// return success for now
  return(1)

' ////////////////////////////////////////////////////////////////////////////


PUB Flash_Erase( block ) | _index, _spi_data, _flash_start_addr
{{
This function erases a block of FLASH memory.

PARMS:

block - block address to erase.

RETURNS: None.

}}
  '// STEP 1: enable writing to chip again
//////////////////////////////////////////////////
  '// enable SPI interface

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, WRITE_ENABLE, $FF )

  '// disable SPI interface

  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )
  Delay_MS( 100 )

  '// STEP 2: erase 4K block of chip
  _flash_start_addr := FLASH_BLOCK_TO_ADDR ( block )

  '// enable SPI interface

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, BLOCK_ERASE4, $FF )

  '// write block address, there are only a handful of blocks since each is 4-64K
  _spi_data := SPI_Write_Read( 8, _flash_start_addr >> 16, $FF )
  _spi_data := SPI_Write_Read( 8, _flash_start_addr >> 8, $FF )
  _spi_data := SPI_Write_Read( 8, _flash_start_addr >> 0, $FF )

  '// disable SPI interface
  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )

  '// erasure takes a long time...
  Delay_MS( 500 )

  '// return success for now
  return(1)

' // end Flash_Erase
```

```
' ////////////////////////////////////////////////////////////////////////////

PUB Flash_Write( flash_start_addr, buffer_ptr, num_bytes) | _index, _spi_data
{{
This function writes num_bytes bytes to the FLASH memory at flash_start_addr from the sent
buffer_ptr.

It uses the byte/page write function. Although, there is a "sequential" byte write function
available on the AT26DF081A the new rev of the Atel chips (AT25xxxx series) coming out in
2011 have replaced the sequential write with another function :(, thus we simply have to
check for page boundaries and re-issue another page write every 256 bytes.

PARMS:

flash_start_addr - starting address in the FLASH memory to write the data at.
buffer_ptr       - local byte pointer to Prop memory holding data to write to FLASH memory.
num_bytes        - number of bytes to write to FLASH from source buffer_ptr.

}}

  '// STEP 1: enable writing to the chip again
  '// enable SPI interface

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )
  _spi_data := SPI_Write_Read( 8, WRITE_ENABLE, $FF )

  '// disable SPI interface

  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )

  '// STEP 2: write buffer to address in flash //////////////////

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )

  '// initiate byte/page program command
  _spi_data := SPI_Write_Read( 8, BYTE_PROGRAM, $FF )

  '// write starting address byte by byte, MSB first...
  _spi_data := SPI_Write_Read( 8, flash_start_addr >> 16, $FF )
  _spi_data := SPI_Write_Read( 8, flash_start_addr >> 8, $FF )
  _spi_data := SPI_Write_Read( 8, flash_start_addr >> 0, $FF )

  '// write first byte of
  _spi_data := SPI_Write_Read( 8, byte [buffer_ptr][ 0 ], $FF )

  _index := 1

  '// write remaining bytes
  repeat until (_index => num_bytes )

    ' now test if we are on a page boundary, if so finish page, and start new page
    if ( ((flash_start_addr + _index) // 256) == 0)
      '// set CS to SPI select channel 0 (null)
      SPI_Select_Channel( 0 )

      ' replace this with polling...
      Delay_MS(10)

      '// set CS to SPI select channel 3 (FLASH)
      SPI_Select_Channel( SPI_CHAN_FLASH )
      _spi_data := SPI_Write_Read( 8, WRITE_ENABLE, $FF )
```

```
      SPI_Select_Channel( 0 )

      SPI_Select_Channel( SPI_CHAN_FLASH )

      '// re-issue byte/page program command
      _spi_data := SPI_Write_Read( 8,BYTE_PROGRAM,$FF )

      '// write starting address byte by byte, MSB first...
      _spi_data := SPI_Write_Read( 8, (flash_start_addr + _index) >> 16, $FF )
      _spi_data := SPI_Write_Read( 8, (flash_start_addr + _index) >> 8, $FF )
      _spi_data := SPI_Write_Read( 8, (flash_start_addr + _index) >> 0, $FF )

      ' end if

    '// write the next byte of data
    _spi_data := SPI_Write_Read( 8, byte [buffer_ptr][ _index ], $FF )

    _index++

    ' // end for index

  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )

  ' write final bytes left in internal buffer, replace this with polling...
  Delay_MS(10)

  '// disable SPI interface

  '// return success
  return(1)

' // end Flash_Write

' ////////////////////////////////////////////////////////////////////////////////

PUB Flash_Read( flash_start_addr, buffer_ptr, num_bytes) | _index, _spi_data
{{
This function reads num_bytes from the FLASH memory starting at memory location
flash_start_addr and stores the data in
buffer_ptr.

PARMS:

flash_start_addr - starting address in FLASH memory to read data from.
buffer_ptr       - local Prop byte pointer to store the read data at.
num_bytes        - the number of bytes to read from the FLASH memory.

RETURNS: None.
}}

  '// set CS to SPI select channel 3 (FLASH)
  SPI_Select_Channel( SPI_CHAN_FLASH )

  '// initiate read command
  _spi_data := SPI_Write_Read( 8, READ_DATA, $FF )

  '// write starting address byte by byte, MSB first...
  _spi_data := SPI_Write_Read( 8, flash_start_addr >> 16, $FF )
  _spi_data := SPI_Write_Read( 8, flash_start_addr >> 8, $FF )
  _spi_data := SPI_Write_Read( 8, flash_start_addr >> 0, $FF )

  '// now read data
  repeat _index from  0 to num_bytes-1
```

```
    byte [buffer_ptr][_index] := SPI_Write_Read( 8, $00, $FF )

    ' // end for index

  '// disable SPI interface

  '// set CS to SPI select channel 0 (null)
  SPI_Select_Channel( 0 )

  '// return success
  return(1)

' // end Flash_Read

' ////////////////////////////////////////////////////////////////////////////

'////////////////////////////////////////////////////////////////////////////
' FLASH MACROS ///////////////////////////////////////////////////////////////
'////////////////////////////////////////////////////////////////////////////

PUB FLASH_ADDR_TO_BLOCK( n )
  n := (n >> 12)

' ////////////////////////////////////////////////////////////////////////////

PUB FLASH_BLOCK_TO_ADDR( n )
  n := (n << 12)

' ////////////////////////////////////////////////////////////////////////////
```

The API itself is heavily commented and in most cases there are only a few lines of code that perform the same logic over and over just as the other APIs we have seen. The functions build the command word up, send the bytes over the SPI bus, and exit. That's about all there is to the FLASH system; remember you will have to manually include the source, constants and data structures from this demo to use the FLASH API in your applications.

In closing, the idea of the FLASH in the C3 is of course to store data, program code, boot operating systems, data logging and much more. The FLASH chip is always with the C3 (unlike the SD card that can be removed), so it's a great feature which I will leave to you to figure out novel uses.

3.3.12 SD Card Demo

The SD card is one of the most amazing technologies available today. Cards with over 64 GB are available with access speeds in the hundreds of megabytes per second. All that technology comes at a price and that's the complexity of the software to access the cards. Surprisingly, the hardware interfaces are trivial. SD cards use either a serial SPI interface or a parallel 4-bit interface, each no more complex than a few signals and a couple pull-up resistors! The software on the other hand is daunting and not something you want to have to write yourself. If you are a thrill seeker and want to write your own driver from scratch you will have a lot of work to do. First, SD cards have no file system; they are raw devices consisting of a number of sectors (each sector usually **512 bytes** each). This way, any client driver can access them as it wishes and the SD card is not locked into any specific file system or format.

In fact, this is a very misunderstood aspect of SD cards. Many people think there is a file system on the SD card that is native. There is not, we simply write data to the sectors using conventions that map into FAT16, FAT32, NTFS, Ext2, Ext3, etc. So, to write a complete SD card file system you need the following layers:

- SPI driver to send and receive bytes to the SD card.
- A low-level driver to communicate with the SD card and access sectors; read, write, erase, using SD card protocol.
- A mid-level driver that can access any sector and read or write it. This is the basis for any modern file system.
- A high-level file system driver that formats the SD card by writing to the sectors and constructing a file system that mimics the standard file systems found on PCs; FAT16, FAT32, Ext2, etc.
- A higher-level API that allows you to create files, read, write, format, create directories and so forth.

If this sounds like a lot of work—it is! More or less, you are talking about a complete file system from the ground up like DOS. Alas, SD card drivers are far and few between on many microcontrollers including the Propeller. Considering this, the SD card support in the C3 is based on the simplest interface for SD cards: the SPI interface. This allows the easiest porting of drivers to the C3 from other authors. Currently, there are a couple SD card drivers for the Propeller chip that I know of and these have actually been ported to the C3 by other coders. The problem with any other SD card driver is that most assume a simple single IO pin to assert or de-assert the chip select on the SD card. However, the C3 has an SPI bus decoder that we need to call a function to select the correct SPI device (SD card in this case). Thus, these other Propeller SD card drivers need a little hacking to add the few lines of code to make the chip select logic work on the C3 properly, which other open-source coders have done. You can find these drivers and other applications in the FTP site here:

PropC3 \ Apps \ *.*

However, to make things easy I have written one as well to get you started that is 100% Spin based, so you can follow along. The SD card demo is different from any of the other demos in that it doesn't use the same template as the other demos. It is such a complex program that I made it from scratch along with the driver which is named c3_sd_drv_010.spin. This is the only external driver I made for the C3. All the other driver APIs are simple enough that I prefer you copy source code and build your own from a few lines of code. But, the SD card is a monster and needs a driver. So, for any SD card stuff you need to include the driver c3_sd_drv_010.spin which is found here on the FTP site:

PropC3 \ Sources \ c3_sd_drv_010.spin

The driver is heavily commented and has built in documentation like a small manual or book. Also, the Hardware section on the SD card interface in Section 2.13 has a number of document files you should read up on about SD cards protocol, FAT file systems and so forth. Moving on, I use my driver to create the demo for this part of the manual. The demo's name is c3_sd_demo_010.spin and it too is located on the FTP site here:

PropC3 \ Sources \ c3_sd_demo_010.spin

As I said, the driver is documented internally, so please read that, but one thing you will notice is that there are video terminal functions in it? This is for debugging purposes; when you don't need them, you can comment out all mention of them and pull the included sub-object. But, for now, don't worry about it; just use the driver and the demo to get started. So, let's take a look at the demo itself. Go ahead and build the demo program, download the C3 and run it. You should see something like that shown in Figure 3.14 below (press any key after the intro screen).

Figure 3.14 — SD card demo main menu.

The main menu has a number of advanced options, but the ones we are interested in immediately are:

- (1) Mounts/Re-mounts the SD card.
- (5) Prints a directory of files on the SD card.
- (6) Loads and prints a file to the screen.

At this point, we have a little problem: no SD card is inserted. Therefore, if you have a microSD card you need to format it to FAT16 (that's all my driver understands) then go ahead and copy a few text files on the card. And make sure that the filenames are in 8.3 format—this isn't Windows 7, we have to keep it simple!

Once you have an SD card formatted with files on it, go ahead and insert it into the C3, facing up, insert it carefully—do not force it—just give it a push and it should click in. If it seems to bind, just pull it back a bit and re-try. The mechanical interface is brand new and needs to get worked in a bit.

!	**The SD card mechanical interface is spring loaded.** It is called a "push-push" interface because you push the SD card in to insert it until you feel it lock in place, then you push it again until it unlocks at which time you can pull it out. Do not force the card out by pulling on it.

Now that the SD card is in, we are ready to **mount** it. This term probably doesn't mean much if you are a Windows person and have never worked with Unix or Linux. But, mounting simply means that a file system is mounted on a system and the system becomes aware of it and can access it. In our case, this means that the driver will try to access the SD card, initialize it, and scan for some FAT16 markers to make sure that there is a FAT16 file system with a boot sector and partition. So here are the main menu steps at this point to see what's on the SD card:

- Select option (1) to mount the card. If it can't mount the card try again, and or eject the card and re-insert it. To eject the SD card, push it again—do NOT pull it!
- Select option (5) to display the directory of files, you should see all your files on the card.
- Select option (6) to load a file and display it to the screen with a very crude screen print.

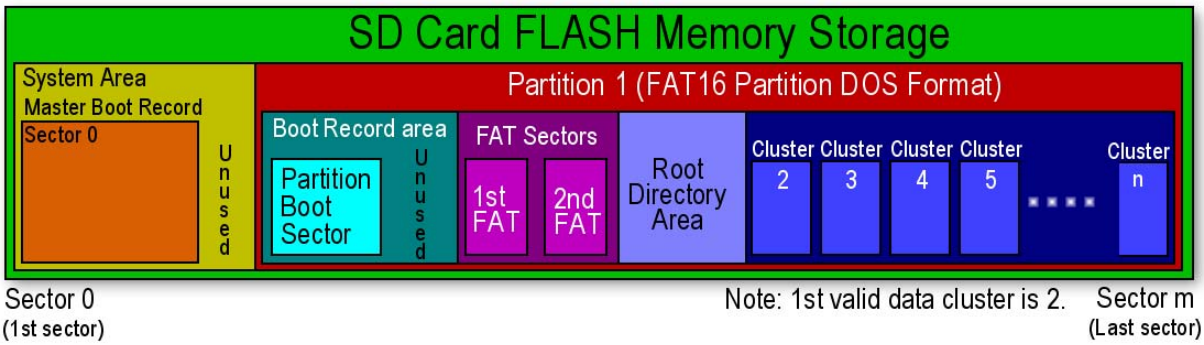
And that's it! The other options are for advanced users that are more familiar with FAT file system mechanics, they print out the master boot record, partition entry 0, and the partition boot record. All of those are described in detail in the HYDRA SD MAX manual referred to in the Hardware section 2.13 on the SD card.

Now that we have seen the demo, let's take a brief look at the API function listing and review that.

SD Card API Listing

The API for the SD card is documented in the driver itself, so please refer to that as well as the demo program as a model of getting an SD application up and running. The SD driver more or less has to handle a complete FAT16 file system which looks something like the illustration shown in Figure 3.14.

Figure 3.15 — FAT16 file system diagram.



As you can see, the FAT16 system mapped into the SD card is complex and has a lot of sections in there. The FAT file system is beyond the scope of this manual, but my other booklet on the HYDRA SD MAX included in the C3 FTP site gives a really good explanation of it all. You can find it here:

PropC3 \ Docs \ HYDRASDMAX \ *.*

The driver developed for the C3 is a simple port of the HYDRA SD MAX driver and took less than 5 minutes to port (simply had to change chip select logic to use the C3 SPI channel selection function). Now, let's take a look at the driver API function list (I have put them into a source code block since there are so many, and its easier to see them at a glance this way):

```
{{
*** Initialization Functions ***

  PUB Start ( buffer_ptrs ) - Initializes the driver and writes the callers pointer storage
  space with local disk buffer addresses


*** SPI Functions ***

  PUB SPI_Init(mode) - Initializes the hardware SPI interface I/O pins from propeller
  to card.

  PUB SPI_Send_Recv_Byte(spi_data8) - Sends a single byte to SPI interface and receives
  one at the same time.

  PUB SPI_Read_Byte - Reads the SPI buffer, writes a dummy values of $FF.

  PUB SPI_Write_Byte(spi_data8) - Writes a byte to the SPI interface.


*** SD Functions ***

  PUB SD_Mount - Mounts the SD card by initializing and placing it into SPI mode.

  PUB SD_Unmount - Unmounts the previously mounted SD card.
```

```
  PUB SD_Send_Command( command8, address32 ) - Sends a generic command to the SD card.

  PUB SD_Read_Sector(sector32, sectorbuffer16) - Reads a sector from the SD card.

  PUB SD_Write_Sector(sector32, sectorbuffer16) - Writes a sector to the SD card.

  PUB SD_Wait_Write_Complete - Internal function used in the write command to make sure
  the flash has been updated.

  PUB SD_Print_Sector(sector32, sect_ptr, start_byte, end_byte, base, print_addr)
  - diagnostic function that prints the contents of a sector to terminal.

  PUB SD_Read_WP - reads the single bit "write protect" signal on the SD card mechanical

  PUB SD_Read_CD - reads the single bit "card inserted" signal on the SD card mechanical


*** FAT16 Functions ***

Lower level functions (initialization of FAT16 system)

  PUB FAT_Read_MBR(mbr_ptr) - Reads the MBR (Master Boot Record) from the SD card, sector 0.

  PUB FAT_Load_Partition_Entry(partition, mbr_ptr) - Loads the requested 16-byte partition
  entry from the loaded MBR.

  PUB FAT_Print_Partition_Entry(partition, mbr_ptr) - Pretty prints the partition entry to
  terminal.

  PUB FAT_Load_Partition_Boot_Rec(pbr_ptr) - Loads the partition boot record referred to by
  loaded partition entry.

  PUB FAT_Print_Partition_Boot_Rec - Pretty prints the partition boot record (aka volume
  record or simply boot record).

*** High level functions (general file I/O) ***

  PUB FAT_Print_Directory - Print the root file directory to the terminal (like DOS DIR
  command).

  PUB FAT_File_Open(filename_ptr, file_handle_ptr) - Opens filename and fills in the sent
  file handle structure.

  PUB FAT_File_Close(file_handle_ptr) - Closes the file handle.

  PUB FAT_File_Read(file_handle_ptr, buffer_ptr, count) - Reads bytes from file referred to
  by the sent file handle.

  PUB FAT_File_Seek(file_handle_ptr, count, mode) - Seeks file pointer to a specific location
  in file for reading.

*** Utility Functions ***

  PUB To_ASCII(inchar, replace_char) - Used to map non-printable characters to printable.

  PUB itoa(value, sptr) - Converts an integer to an NULL terminated ASCII string.

  PUB ToUpper(char) - Converts lower case ASCII to upper case.

  PUB Strncomp(string_ptr1, string_ptr2, length) - Compares strings.

  PUB _Min(a,b) - returns the min.
```

```
  PUB _Max(a,b) - returns the max.

*** Exported Object Functions ***

  PUB tvt_bin( value, digits ) - Prints a binary number to the terminal with specific number
  of digits.

  PUB tvt_out( char ) - Outputs a single character to terminal.

  PUB tvt_dec( value ) - Prints a number in decimal format to terminal.

  PUB tvt_hex( value, digits ) - Prints a hex number to the terminal with specific number
  of digits.

  PUB tvt_setx( new_x ) - Sets the x cursor position on terminal.

  PUB tvt_sety( new_y ) - Sets the y cursor position on terminal.

  PUB tvt_getx - Gets the current x cursor position in terminal.

  PUB tvt_gety - Gets the current y cursor position in terminal.

  PUB tvt_pstring( string_ptr ) - Prints a NULL terminated string to the terminal.

  PUB tvt_erase - Erases completely the character under the current cursor position.
}}
```

Comments on the API Functions

- Make sure to refer to the return value from each function; the functions have non-homogeneous return codes. For example, in some cases 0 is an error, in some cases it means success.
- Many of the driver functions have sprinkled terminal print statements that are left in them in case you want to output more debug information. If you don't want any NTSC screen printing support simply remove all references to term.* calls in the code and don't include the terminal driver.
- The functions are written for maximum clarity, not speed. I suggest re-writing everything in ASM, but these Spin versions are reasonably easy to follow, so you can learn the ropes of SPI, SD, FAT16 drivers.
- The FAT16 driver layer is minimal and doesn't support writing to files, and only supports the root directory and no folders. But, that should suffice for 99% of applications. Adding file writing isn't hard, but it does mean a lot of work allocating clusters and setting up links. These operations are easily done using the PC to create the file. However, if you need data logging, obviously you will need to create and write a file, so you will have to add this function yourself.
- Everything has been tested, but don't assume everything is working perfectly. If something doesn't work as expected test it thoroughly, but it could very well be the driver. They were only tested functionally, but not in any large application, so they are not threshed out and probably have bugs here and there. But, nonetheless they are good for tutorial use and to get you started, so you don't have to write your own!
- Remember, ***always*** format your SD cards FAT16, and make sure you do ***not*** use folders and all your file names are ALWAYS 8.3. Also, it's suggested that you use 1–2 G SD cards, these seem to work the best.

There are a lot of microSD card manufacturers out there, and driver authors sometimes do not follow the SD card specification 100%. This causes problems with various cards. Open source authors tend to do things on a budget and that means that they don't thoroughly test things with a lot of different hardware. In the case of SD cards, a professional piece of software sold would require testing that included at least 20–30 SD cards from various manufacturers. I personally tried my driver on 10 SD cards, and it works on them all. However, I have tried other drivers from other authors that will only work on Kingston or Sandisk media for example. So, point is, it can't hurt to buy a couple SD cards. I suggest Kingston and Sandisk or other name brands. Try to buy online as well; stores tend to charge 3–5 times what you should pay. Some of my favorite sources are Newegg.com and Dz tech.com for really low prices.

3.4 Serial Version Demos (Using USB UART)

All the demos in the previous section used the C3's local PS/2 keyboard as the input device. For fun, I ported all of the demos to use the Parallax Serial Terminal program instead. This is of course redundant and not necessary, but a lot of customers really like talking to the Propeller dev boards via a serial connection and not having to use two keyboards for input. That alone is enough motivation to make the ports. That said, this section will be very short and only give the names of the demos and any other details that might be of interest. The API listing, logic, and other technical details are all the same as in the previous section, so look there if you have skimmed to this section and are jumping around the manual.

3.4.1 Setting up for the Demos

Now, for the tricky part. The Propeller Tool IDE requires the use of the USB serial port for programming, so obviously you can't use the Propeller Serial Terminal and the Propeller Tool IDE at the same time! Therefore, with the Propeller Serial tool, or any serial terminal, you need to disconnect it first from the serial port then program the C3, then re-connect it. This is a pain, but only a couple mouse clicks.

If you aren't using the Parallax Serial Terminal then you will have to make sure to disconnect your terminal program before making the connection to the C3 with the Propeller Tool, otherwise it will complain. The Parallax Serial Terminal is included with the Propeller Tool software v1.2.7 (launch from the Start menu) and it can be downloaded from the Parallax main site at:

http://www.parallax.com/propeller

Click on the Propeller Downloads link and find the latest Parallax Serial Terminal.

Whether you use the Parallax tool or another one (ZOC, Putty, etc.) you should set your serial terminal to **38,400-N81**:

Baud:	38400
Parity Bit:	N
Data Bits:	8
Stop Bits:	1
Echo:	Local echo off (check box on the Parallax Serial Terminal next to the **<Pref's>** button).

Once the serial tool is running, put it next to the Propeller Tool IDE and run them side by side, so you can quickly enable/disable the serial tool to release and engage it from the serial port as needed. The process for each demo will be the following:

Step 1. Load demo top level file into Propeller Tool. All the demos have nearly identical names to the stand alone version, but with "sterm" appended to the file name. It stands for "serial terminal" version.

Step 2. Build the program and download it to the C3 by pressing F11 or selecting the appropriate menu option with the mouse.

Step 3. Re-connect the serial terminal to the correct COM port at 38400 N81.

Step 4. Hit the RESET button on the C3 to re-establish connection with the serial terminal.

Step 5. Play with the demo.

All of the demos will boot the C3's NTSC video hardware and display the C3 logo screen. It's not necessary to plug in an NTSC monitor, but can't hurt, so you can confirm the C3 is running perfectly.

The demos will print to the serial terminal as they did the NTSC screen in the previous section. The only difference is the editing might be slightly different depending on your terminal software.

3.4.2 Port A/B Demo

The file name for the serial version of the port IO demo is **c3_port_io_demo_sterm_010.spin**, it's located on the FTP site here:

PropC3 \ Sources \ c3_port_io_demo_sterm_010.spin

3.4.3 NES Gamepad Demo

The serial version of the NES gamepad demo is **c3_nes_gamepad_demo_sterm_010.spin**, it's located on the FTP site here:

PropC3 \ Sources \ c3_nes_gamepad_demo_sterm_010.spin

3.4.4 Servo Port Demo

The file name for the serial version of the servo demo is **c3_servo_demo_sterm_010.spin**, it's located on the FTP site here:

PropC3 \ Sources \ c3_servo_demo_sterm_010.spin

3.4.5 SRAM Demo

The file name for the serial version of the SRAM demo is **c3_sram_demo_sterm_010.spin**, it's located on the FTP site here:

PropC3 \ Sources \ c3_sram_demo_sterm_010.spin

3.4.6 A/D Demo

The serial version of the analog to digital demo is **c3_analog_digital_demo_sterm_010.spin**, it's located on the FTP site here:

PropC3 \ Sources \ c3_analog_digital_demo_sterm_010.spin

3.4.7 A/D Plus SRAM Demo

The serial version of the hybrid A/D plus SRAM demo is **c3_ad_sram_demo_sterm_010.spin**, it's located on the FTP site here:

PropC3 \ Sources \ c3_ad_sram_demo_sterm_010.spin

3.4.8 FLASH Memory Demo

The file name for the serial version of the FLASH demo is **c3_flash_demo_sterm_010.spin**, it's located on the FTP site here:

PropC3 \ Sources \ c3_flash_demo_sterm_010.spin

4 Porting Applications from other Boards to C3

The Propeller chip was released in 2006 with only two application boards: the HYDRA and the Propeller Demo Board. Since then, dozens of new Propeller-based boards have been created. The good news is that most rely on the Propeller chip itself for 90% of the functionality, so the same bus configurations, IO pins and drivers are used for the majority of these designs. Therefore, in this section, I am going to give you some basic tips on porting applications from other boards to the C3. Please refer to the Hardware portion of this manual for specifics on pin outs and electrical designs.

4.1 Propeller Chip, Reset, and Clocking

The Propeller chip on the C3 uses an external crystal like most designs. The crystal itself is socketed, so you change it. Currently the popular speeds are 5, 6.25, and 10 MHz. The 5 and 10 MHz are used for normal clocking with PLL multipliers of 16 and 8 respectively (resulting in an 80 MHz system clock). The 6.25 MHz crystal is used to over-clock the Propeller to 6.25 MHz x16 PLL = 100 MHz. So, if you are porting a piece of software from another platform to the C3, make sure you either adjust the clock rate lines in the code or change the crystal. The clock rate is usually found at the very top of the main source file.

The reset system on the C3 is also very similar to most designs. The only interesting thing is that an external device can reset the Propeller via the expansion IO headers by pulling the **RESn** line low (System/Power header) and releasing it again.

4.2 Porting VGA Drivers

The VGA on the C3 uses IO pins P23..P16. This is one of the most common pin groups used on Propeller products for VGA. If you have a board that is using a different pin group and you want to port a VGA driver then you need to find the code that controls the pin group and make sure to change it P23..P16. Additionally, make sure you enable the IOs to outputs for the same pins, P23..P16!

The second thing you need to do in all cases is add a couple lines of code that enable the VGA buffer, so the signals from the Propeller make it to the VGA header. This is accomplished with the following code fragment:

```
DIRA[STATUS_LED_BUS_MUX] := 1 ' set to output
OUTA[STATUS_LED_BUS_MUX] := 0 ' turn on buffer
```

4.3 Porting Composite Video Drivers

Composite video is a bit tricky since it's a more complex setup and you really have to know what you're doing. But, similar to porting VGA drivers you need to locate in the driver where the pin group is set to and change it to the C3's pin group for video which is P15..P8, the second group. But, there is a catch—the composite video uses only 4 pins for the DAC, so you have to tell it which nibble and set baseband or broadcast as well. So, first, you need to adjust the nibble to the upper nibble bits P15…P12. And you need to adjust any bitmasks that are used by the driver, so they look like %1110_0000. Also make sure you set P15..P13 as outputs in the driver. Finally, note that P12 is actually not used by video on the C3, it controls the VGA buffer. This is acceptable since the last bit in composite video setups is used for aural signals for broadcast which we don't use.

So, in review to port a composite video driver you must:

- Change the IO pins to P15..P13.
- Set them to output.
- Search the driver and make sure the pin group is set to the 2nd pin group (they are counted in 8's, 0 based, so the C3 is pin group 1 numerically, pin group 2 physically).
- Find any bitmasks and make sure they refer to the proper bits (upper nibble) or if there is a 32-bit bitmask make sure it looks like:

  %0000_0000_0000_0000_1110_0000_0000_0000

  ……or the inverse as appropriate.
- Find the baseband / broadcast setting and make sure you have it set for upper nibble baseband.

4.4 Porting Audio Drivers

Audio on the C3 uses a single IO pin (**AUDIO_MONO** at P24) standard PWM setup with a low-pass integrator filter with 3 db point at 1 kHz. The majority of Propeller designs use this kind of setup. The only thing that you might have to do to port another sound driver is change the pin assignment and if there is stereo output, disable the other pin, or mix in software first and then send the sum to the single mono output on the C3.

4.5 Porting PS/2 Drivers

The C3 uses a 2-signal PS/2 interface comprised of PS/2 data (P26) and clock (P27), so all you need to do is change your driver to these pins and make sure to use a 2-pin driver not the 4-pin driver.

4.6 Porting SD Card Drivers

All SD card drivers use an SPI driver of some sort, so the first step is to make changes to the MISO (P11), MOSI (P9), and SCK (P11) signals to match that of the C3. Then you must find the code that asserts the chip select (CS line) on the SD card and change it from a simple IO high/low to a call to the SPI system channel select function **SPI_Select_Channel (..)** covered previously. Also, make sure to include the SPI bus functions and to initialize the SPI bus.

4.7 Supporting NES Controllers for Games

The C3 obviously has no NES ports on it, so the only way to support NES controllers is to buy (or make) an NES adapter that plugs into the expansion header port and is powered from the port as well as signaled from it. The NES drivers usually consist of only three signals: clock, data, and latch. You must make sure to assign these properly to wherever your particular NES arrangement places them in IO space.

4.8 USB Serial Considerations

The C3 uses the FTDI 232R USB to serial UART which more or less does everything. As with other Propeller designs the DTR signal is connected to reset via a capacitor and a NPN transistor that grounds (resets) the system if it senses a DTR pulse.

4.9 EEPROM Support

The C3 currently ships with a 64 KB EEPROM. If you have an application that requires a 128 KB EEPROM, you are out of luck. But, that's what the 1 MB FLASH is for. Of course, if you really want 128 KB, you can always de-solder the EEPROM on the C3 – I suggest using Chip Quick™ if you are hell-bent on upgrading to a larger EEPROM and want to remove the current one (or a good hot air de-soldering gun).

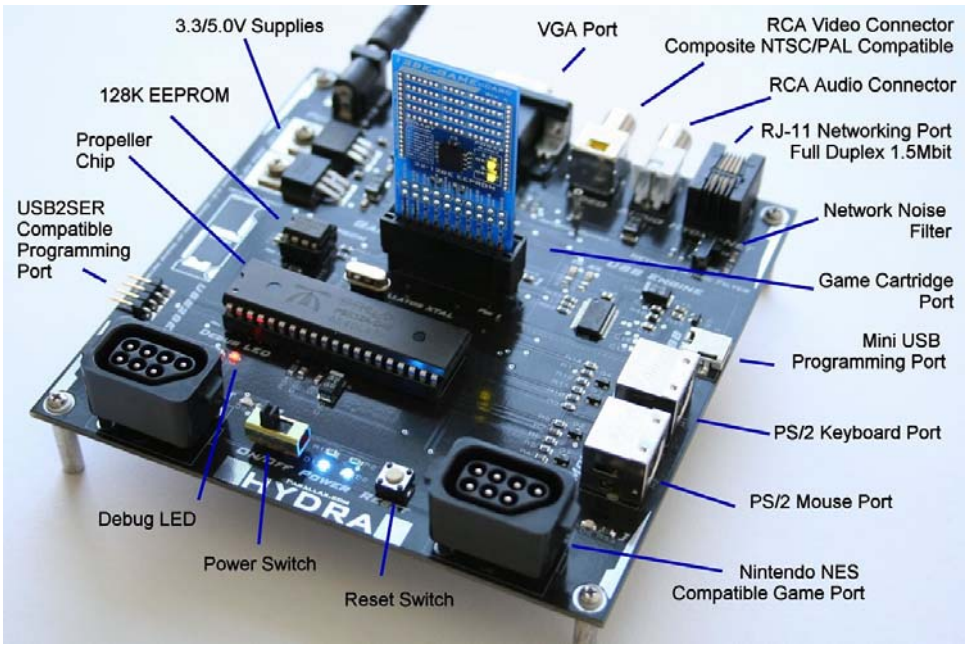
4.10 Power Supplies

The C3 has hearty supplies that can source 1.5 A. However, when sourcing this much current the regulators will get very hot and burn your fingers, so don't touch the heat sink. The other detail about the C3 is it can be USB powered as long as you don't draw more than 500 mA.

4.11 Porting HYDRA Applications

The HYDRA and the C3 share many of the same IO mappings and features while some features are radically different and others are non-existent. In this section, we will review some of the differences to help you port applications you (or others) have written for the HYDRA to the C3 with respect to each sub-system. Below is a quick reference list you can use as you port an application to help you quickly identify the differences between systems. This conversion list is from HYDRA to C3, thus, I will only mention key systems that are in common.

Figure 4.1 — The HYDRA Game Console Rev A



4.11.1 Propeller Chip, Reset, EEPROM, and Clock

The Propeller chip and reset are identical on both systems. However, the clock crystal on the HYDRA is at 10 Mhz, while the C3 is at 5 MHz. Thus, you need to make sure to change the clock frequency setting on HYDRA ports (or use a 10 MHz crystal in the C3). On the EEPROM front, the HYDRA has a 128 KB EEPROM while the C3 has a 64 KB EEPROM, so any apps that require the larger 128 KB will have problems if they use the other 64K.

4.11.2 Composite Video

The HYDRA actually has all four video lines connected to the video DAC, but 99% of all applications only use the standard baseband video lower three lines. Other than that the C3 has the video DAC on a different set of IOs as shown below:

HYDRA Video DAC:	P24..P27 (lower nibble of pin group 3).
C3 Video DAC:	P12..P14,P15[1] (upper nibble of pin group 1).

Note 1: P15 used for **STATUS_LED_BUS_MUX** signal (controls VGA enable and status LED).

4.11.3 Audio

Audio is usually used with the composite video signal, but it can be a stand-alone signal or maybe used with VGA. In either case, both audio signals on the HYDRA and C3 are designed to be PWM generated and work well with a 75 Ω terminating resistance like a TV set amplifier. Both use a low-pass integrating filter and an AC coupling capacitor to the audio RCA output. And both have the same 3 db cutoff frequency. The only difference between the HYDRA and C3 are the I/O pins as shown below:

HYDRA Audio Pin:	P7
C3 Audio Pin:	P24

4.11.4 VGA

Luckily there is less flexibility with VGA than there is with the NTSC DAC since VGA has no upper/lower nibble select, no baseband or broadcast. VGA simply works on one of the 4 pin groups; 0..3, simple as that. The good news is both the HYDRA and C3 use P16..P23 for VGA! Additionally, both systems have a VGA buffer. However, on the HYDRA it's manually controlled with a switch, but on the C3, you control it with the **STATUS_LED_BUS_MUX** signal by asserting it low. So, to port any VGA driver from the HYDRA to the C3, just assert that signal, the VGA pins are the same.

4.11.5 PS/2 Port(s)

Here's where the HYDRA and the C3 diverge quite a bit. The HYDRA uses 4-pin PS/2 ports and has two of them while the C3 only has a single PS/2 port and uses only 2 signals per port. Thus, any game or application that uses the PS/2 ports on the HYDRA must be modified since the C3 can only support keyboard or mouse not both at the same time. Moreover, you need to use a driver that works with 2-signal PS/2 ports like the reference drivers **keyboard_010.spin** or **mouse_010.spin** which can be found on the FTP site here:

PropC3 \ Sources \ keyboard_010.spin
PropC3 \ Sources \ mouse_010.spin

Therefore, you don't really need to port PS/2 drivers from the HYDRA to C3, you just replace the driver with the above or similar and then hook it up to PS/2 pins P26, P27.

4.11.6 HYDRA Game Ports

I have taken the time to port a number of HYDRA games and demos that shipped with the HYDRA itself to the C3. You can find them in the FTP site here:

PropC3 \ Sources \ Games \ HYDRA_Ports \ *.*

Figure 4.2 — The Propeller Demo Board Rev G

4.12 Porting Parallax Propeller Demo Board Applications

The Demo Board and the C3 share many of the same IO mappings and features while some features are radically different and others are non-existent. In this section, we will review some of the differences to help you port applications you might have written for the demo board to the C3 with respect to each sub-system. Below is a quick reference list you can use as you port an application to help you quickly identify the differences between systems. This conversion list is from demo board to C3, thus, I will only mention key systems that are in common.

4.12.1 Propeller Chip, Reset, EEPROM and Clock

The Propeller chip, reset, and clock crystal are identical on both systems. However, the C3 has a 64 KB EEPROM and the Demo Board EEPROM only has 32 KB, but this is fine since the C3 has ***more*** memory, so all EEPROM applications will work without modification.

4.12.2 Composite Video

The Demo Board (like the HYDRA) actually has all four video lines connected to the video DAC, but 99% of all applications only use the standard baseband video lower three lines. The good news is the Demo Board and C3 use the same composite video DAC lines as shown below:

Demo Board Video DAC:	P12..P15 (upper nibble of pin group 1).
C3 Video DAC:	P12..P14,P15[1] (upper nibble of pin group 1).

Note 1: P15 used for **STATUS_LED_BUS_MUX** signal (controls VGA enable and status LED).

4.12.3 Audio

Audio is usually used with the composite video signal, but it can be a stand-alone signal or maybe used with VGA. In either case, both audio signals on the Demo Board and C3 are designed to be PWM generated, but the demo board is routed into an audio amplifier for listening through a set of headphones while the C3 is designed to be output into a 75 Ω terminating resistance like a TV set amplifier. Both use a low-pass integrating filter and an AC coupling capacitor to the audio RCA output. And both have the same 3 db cutoff frequency. However, the C3 will pass frequencies lower than the Demo Board, so you can have really low bass signals pass. Finally, the Demo Board has 2 channels of audio going to the amplifier, the C3 only has one. So, if you have a Demo Board audio application, the

driver can stay the same, but you need to disable one channel; only enable one and make sure it goes to the C3 audio pin as shown below:

Demo Board Audio Pins:	**P10, P11**
C3 Audio Pin:	**P24**

4.12.4 VGA

Luckily there is less flexibility with VGA than there is with the NTSC DAC, since VGA has no upper/lower nibble select, no baseband or broadcast. VGA simply works on one of the 4 pin groups; 0..3, simple as that. The good news is both the Demo Board and C3 use P16..P23 for VGA! But, unlike the C3, the Demo Board doesn't multiplex the VGA signals to other IO pins (it does send them to LEDs though), so any Demo Board VGA driver will work perfectly. The only thing you need to do is assert the **STATUS_LED_BUS_MUX** signal on the C3 by setting it low to enable the C3's VGA buffer.

4.12.5 PS/2 Port(s)

Here's where the Demo Board and the C3 diverge quite a bit. The Demo Board uses 2-pin PS/2 ports and has two of them while the C3 only has a single PS/2 port, but uses the same 2 signals. Thus, any game or application that uses both PS/2 ports on the Demo Board needs to be modified since the C3 can only support keyboard or mouse, but not both at the same time. Nonetheless, they both use the same driver per PS/2 port, unless the Demo Board driver has a combined PS/2 mouse+keyboard driver. In that case you have to completely disable one of the ports since the C3 physically only has a single PS/2 port. Either way, both systems can use the same drivers: **keyboard_010.spin** and **mouse_010.spin** which can be found on the FTP site here:

PropC3 \ Sources \ keyboard_010.spin
PropC3 \ Sources \ mouse_010.spin

Therefore, you don't really need to port PS/2 drivers from the Demo Board to C3, you can just replace the driver with the above or similar (if the driver is a combined keyboard and mouse) and then hook it up to PS/2 pins P26, P27 on the C3. Note: the PS/2 labeled keyboard on the Demo Board uses P26, P27 as well, so if an application on the Demo Board only uses the single PS/2 port it will work without change.

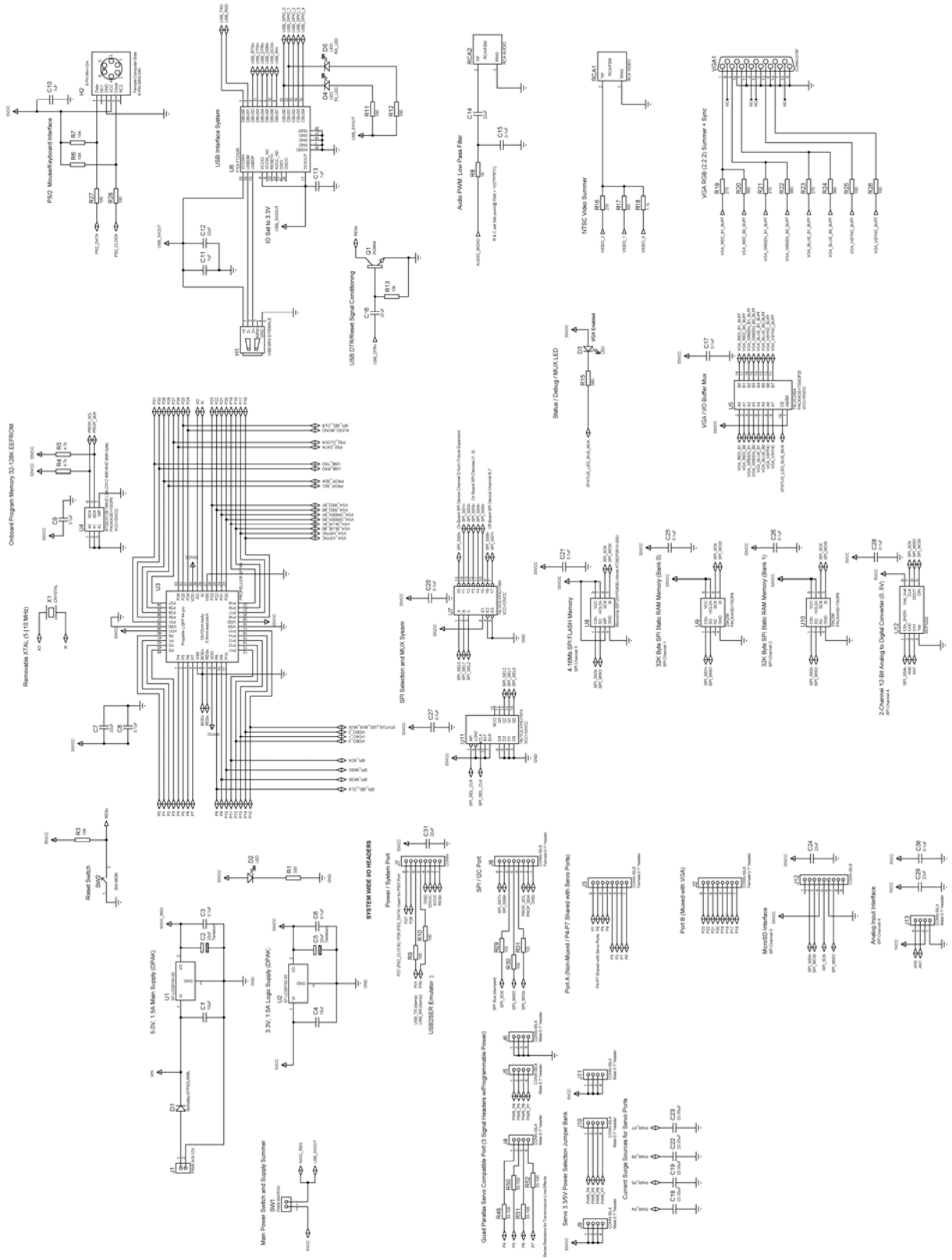
5 Summary

Hopefully this manual has got you off to a good start with the C3. The important thing to remember is that at its core the C3 is a Propeller chip, but with all the peripherals it's a complete computer. Hopefully you and others find new and amazing uses for the C3 and the Propeller chip in this new configuration! Good luck and make sure to check out the Parallax C3 FTP site, it's constantly being updated as are the Parallax C3 and Propeller forums with members just like you posting new projects, answering questions, and exchanging ideas.

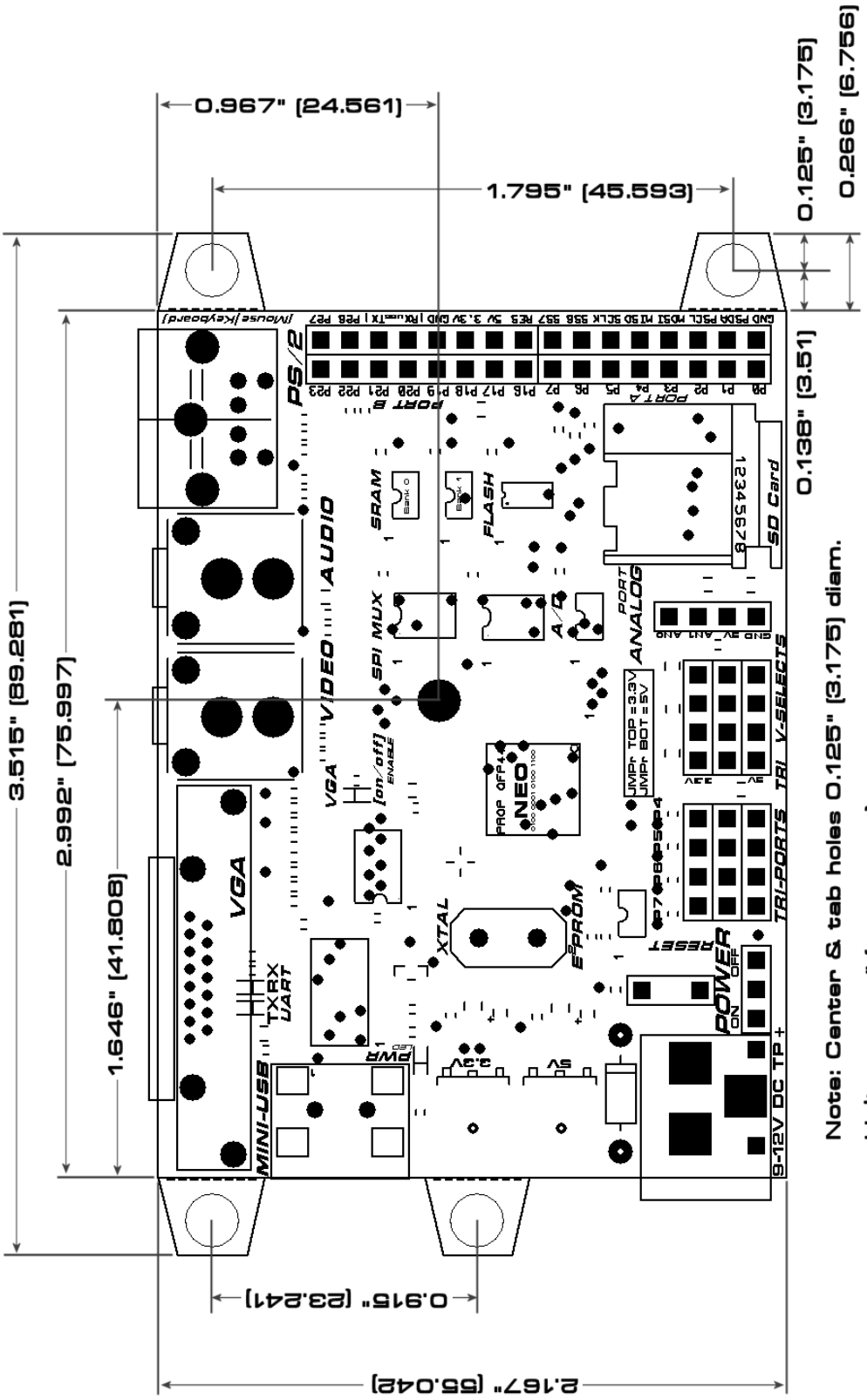
6 Appendix

In this section you will find a few convenient references for the Propeller C3: higher resolution schematics to zoom in on, PCB gerbers, mechanical images, as well as IO assignment at a glance, and FTP site structure.

6.1 Propeller C3 System Schematic



6.2 PCB Mechanical Layout w/Dimensions



6.3 Gerber Images

Figure 6.1 — Top Copper

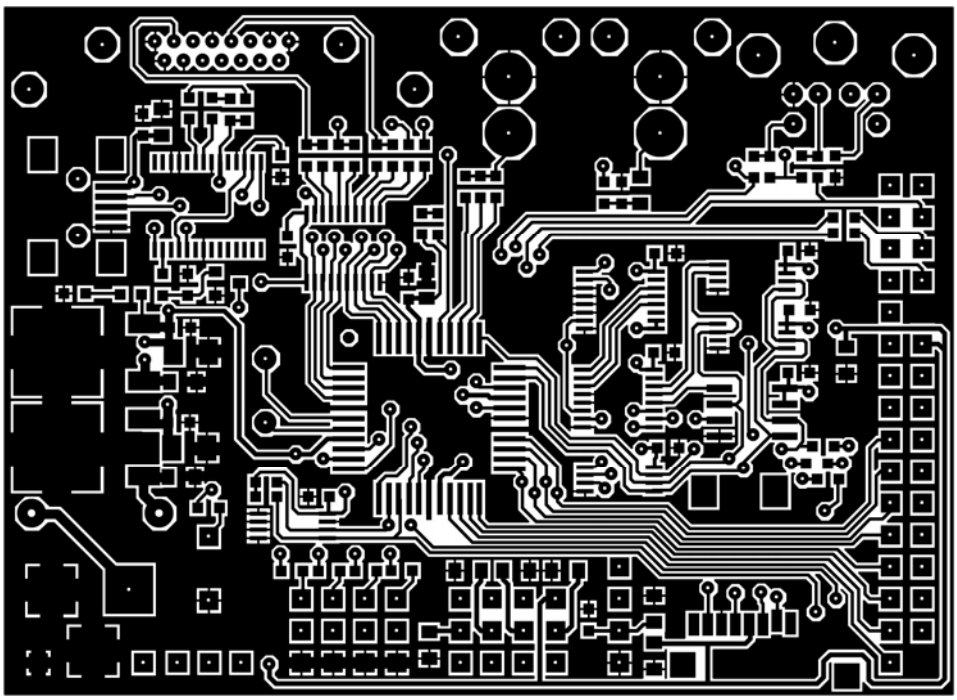
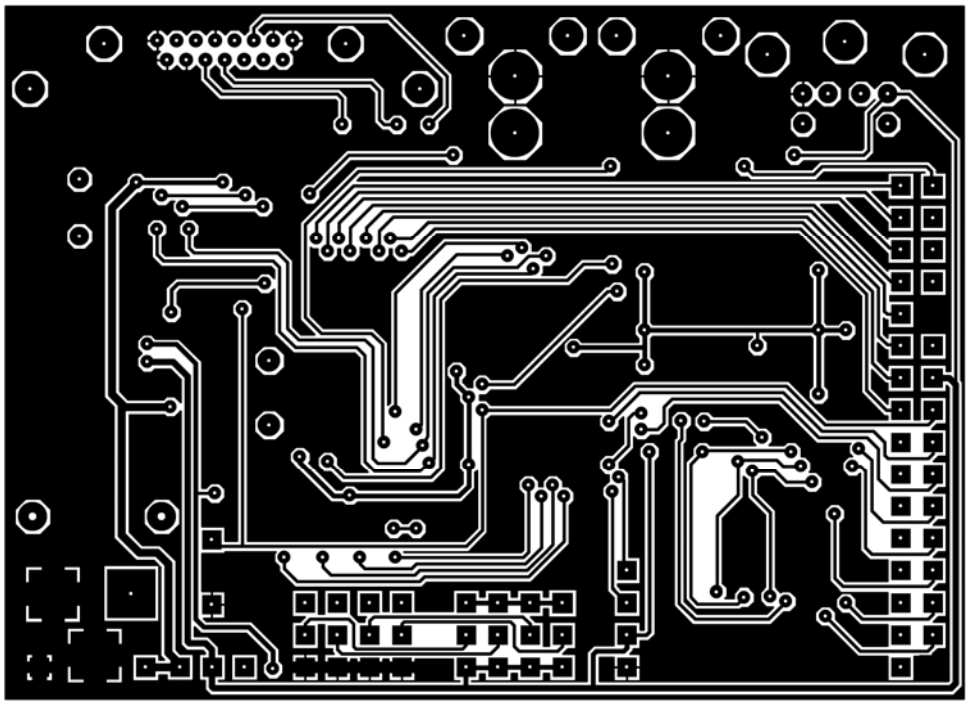
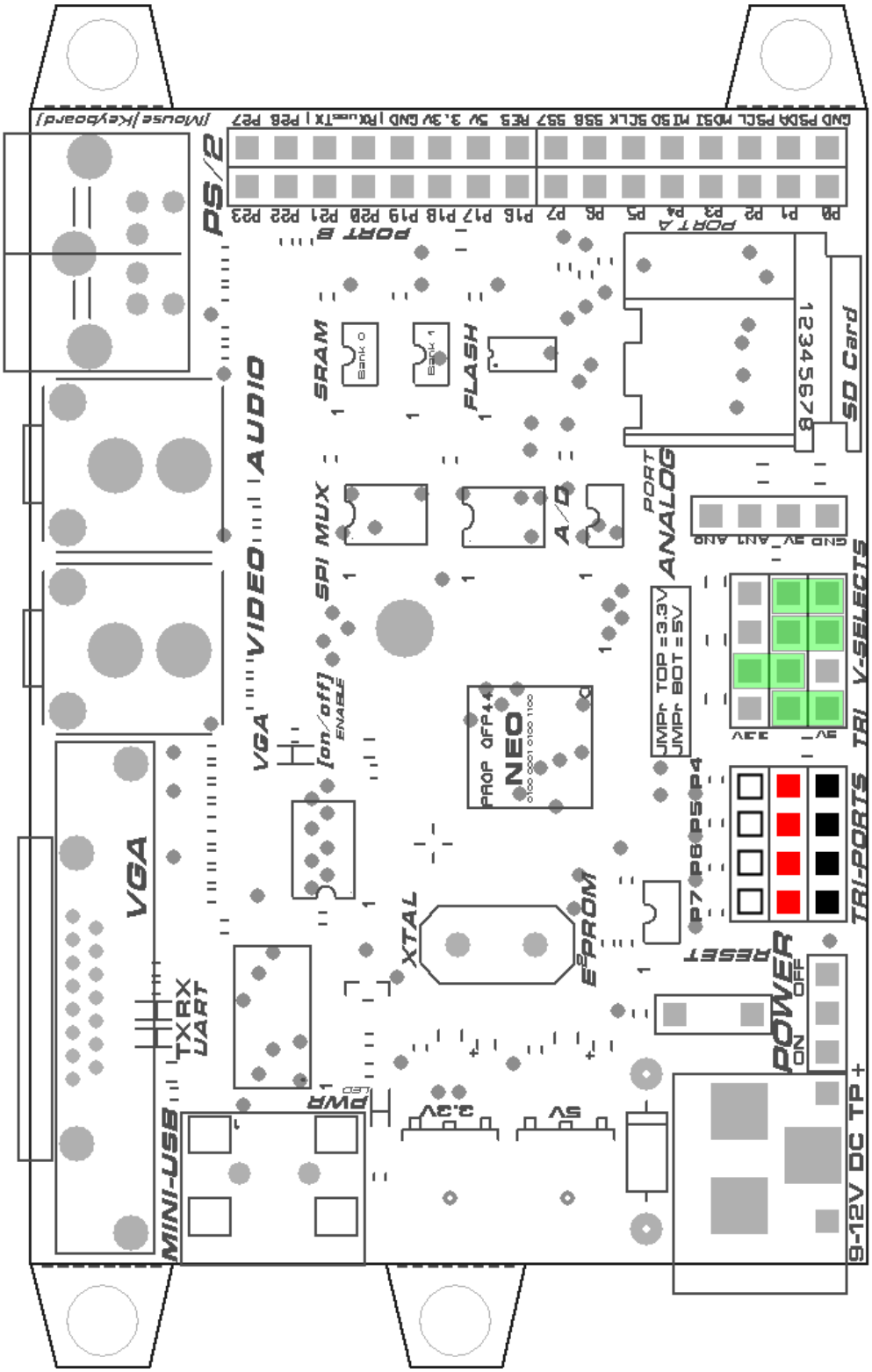


Figure 6.2 — Bottom Copper



6.4 IO Header Pin out Close-up



6.5 FTP Site Layout

All the files for the C3 are located on the Parallax FTP server located here:

ftp:\\ftp.propeller-chip.com\PropC3

The directory structure is shown below:

PropC3 - Root directory.
- **Docs** - Contains documents relating to the C3, datasheets, etc.
- **Sources** - Contains source code and examples from this manual.
- **Games** - Contains games that have been ported or originals.
- **Apps** - Contains applications, languages, and other apps for the C3.
- **Designs** - Contains designs for the C3 including schematics and gerbers.
- **Tools** - Contains tools and programs for the C3.
- **Media** - Contains any extra media for the C3 or videos, audio, etc.
- **Goodies** - Contains any goodies that are special.
- **UPDATE_LOG.TXT** – Changes to the FTP directory are logged here.

Be sure to check out the **Games** and **Apps** directories they contain all of the software written by other authors, ports, etc. The **Sources** directory is the primary source for this manual and the tutorials, but the **Games** and **Apps** have all the good stuff!