Propeller
Table Of Contents

(Hss) Hydra Sound System

The hydra sound system is a realtime audio synthesis playback engine which was created by Andrew Arsenault for use with the propeller.

It allows rapid integration of audio into a propeller based design or program with an easy to use high-level interface.
Simple high-level commands can be issued to (Hss) to control more complex tasks like music playback, sound synthesis and sample playback.

At the moment this object supports a number of propeller based boards (Propeller Demo Board, HYBRID Development Kit and HYDRA Game Console).

(Hss) Supports:

- Four channel tracker music playback engine.
- Dual sound FX syhthesis channels.
- Single 1bit ADPCM channel for compressed sample playback.

(Hss) Supports the following custom file formats:

- (.hmus) 4 channel tracker music format.
- (.hwav) 1bit ADPCM sample file.
- (.snd) uncompressed 8bit sample data.
- (.4snd) uncompressed 4bit sample data.
- (FXsynth Data) small embedded sound FX data.

More information can be found at the (Hss) website: http://www.andrewarsenault.com/hss/

This is a text video driver which was the combined efforts of several forum members.
Potatohead, Hippy, Baggers, Oldbitcollector, and a few others. (Add credit if do)

AiGeneric supports 40x24, 16 color text with multiple fonts, or .64c fonts.
It was created with the idea of being a drop-in replacement for Parallax's text driver.

Package has been divided into two, one supporting the original font system, the other supports loading of .64c fonts.

.start(videopin)
Select video pin to start the driver. (Commonly 12 on Proto/Demo boards)

.str(string("Hello world))
Output a line of text

.center("Hello World")
Centers the words, "Hello World" on the screen

.redefine(65,255,255,255,255,255,255,255,255)
Redefines character 65(A) as a solid block

.color($1A)
Define text color (see demo for common colors)

.out(13)
Output a single character by number (example 13=carriage return)

.hex(255,2)
Display hex number using two digits. (example 255=$FF)

.bin(255)
Display binary number (example 255=11111111)

.dec($FF)
Display decimal number. (example $FF=255)

.cls
Clear the screen

Assembler Subroutines

Because Cog RAM may be modified at run-time, it is not necessary to have a hardware stack to handle subroutine calls within a Cog. A return address for a subroutine call can be placed within a Propeller JMP instruction which will then return execution to after the call. While this simplifies the design of the Cog architecture, it implicitly prevents recursive subroutine calls unless explicit action is taken to prevent return addresses from being overwritten. To use recursive calls will require the user program to implement a software stack for that purpose.

There are two Propeller instructions which are used to implement subroutine calls; **JMPRET** and **JMP**, usually used in their immediate forms (where the bottom 9 bits of the instruction represent the address in Cog RAM of where to jump to).

A **JMP #address** instruction will cause Cog execution to continue at the 'address' specified.

A **JMPRET retAddress,#address** instruction will cause Cog execution to continue at the 'address' specified but will also place the address of the instruction after **JMPRET** into the lower 9-bits of the instruction placed at 'retAddress'. This will normally be a **JMP #address** instruction.

A subroutine call therefore consists of a **JMPRET** which causes execution to continue with the subroutine code which terminates with a **JMP** instruction which will have been modified by the **JMPRET** to return to the address after the **JMPRET** instruction.

To simplify things for the Cog programmer, the Propeller assembler includes two virtual instructions; **CALL** and **RET**. The **RET** instruction allows the **JMP #address** instruction used at the end of the subroutine to be specified without providing an initial address, and the **CALL** instruction allows the programmer to specify just the entry point of the subroutine. In this way the implementation of a subroutine call will appear in the source code much as it would for any other processor instruction set, and the implementation aspects are largely hidden from the programmer.

In order that the **CALL** instruction can determine where the **RET** of the subroutine is it is necessary to give the location of the **RET** a label which is the same as the label of the subroutine with '_ret' appended. For example -

```
Loop        CALL    #MySub
            JMP     #Loop

MySub                               ' Subroutine code here

MySub_ret   RET
```

This is equivalent to -

```
Loop        JMPRET  MySub_ret,#MySub
            JMP     #Loop

MySub                               ' Subroutine code here

MySub_ret   JMP     #0
```

Note that when using **JMPRET**, no "#" is used before the specification of the address which holds the RET instruction.

Nested Subroutines

The Cog architecture does not preclude nested subroutines (where a subroutine can call another subroutine) as each subroutine has its own return address storage associated with it (the **RET** instruction). Only recursive subroutines (where a subroutines calls itself) will present a problem and require a software stack to preserve return addresses to be constructed.

Arbitrary Subroutine Returns

With processor architectures which include a hardware or software stack it is possible to place a return from subroutine instruction wherever it is required and the execution of those instructions will cause an immediate return from the subroutine. Because the Cog architecture does not include a stack and the return address is placed in a single **RET** instruction, a return from subroutine can only be effected by executing that **RET** instruction.

Any arbitrary return from a subroutine must be implemented as a **JMP** to the **RET** instruction. Simply inserting **RET** instructions within a subroutine will not have the desired effect and will cause incorrect operation of the Cog program if executed.

Shared Subroutine Return Points

It is fairly common to have subroutines which call another before returning themselves -

```
Loop        CALL    #Sub1
            JMP     #Loop

Sub1                                ' Subroutine 1 code here

           CALL     #Sub2
```

```
Sub1_ret   RET

Sub2                              ' Subroutine 2 code here

Sub2_ret   RET
```

Such subroutines will often be optimised so the initially called subroutine 'falls-through' into the last call and the return from the subroutine at the end of the last called will act as a return from the subroutine in all cases. Such optimisations may be achieved with the Propeller but it is necessary to label the shared subroutine return point for both possible subroutine calls -

```
Loop        CALL    #Sub1
            JMP     #Loop

Sub1                              ' Subroutine 1 code here

Sub2                              ' Subroutine 2 code here

Sub1_ret
Sub2_ret   RET
```

Assembly Programming

Cog assembler programs are embedded within Spin source code by including them in a DAT section.

While compilation and assembly is underway there are two separate addressing concepts in play; the address of where the assembly code is in Hub memory (and Eeprom and download image) and the address in Cog memory when the assembly code is loaded for execution at runtime. These are very distinct addresses but both are important.

ORG

To ensure that an assembly program's Cog addressing is correct it must begin with an "ORG 0" directive. This ensures that (regardless of where the assembly code may be in Hub memory) Cog addressing is relative to the first instruction which is loaded into the Cog.

When a Cog program is loaded to a Cog from Hub memory, 496 longs of consecutive Hub memory are taken and placed into the Cog. If the Cog program is shorter than 496 longs, whatever follows it in Hub memory will also be placed in the Cog, however, this will be normally be ignored or be irrelevant to the Cog when executing.

Because of this it generally makes no sense to include any ORG directives within the Cog program other than the initial "ORG 0". Doing so will simply create a mismatch between the addressing the assembler maintains during assembly and what is ultimately placed in the Cog. Only if the Cog program itself explicitly relocates what is loaded to where it should be does using ORG make sense.

To ensure that a Cog program will fit within the Cog, the FIT directive may be used. It is recommended that every Cog program ends with a "FIT $1F0" to avoid errors when a Cog program exceeds its largest possible size.

Every assembler instruction of the Cog program will take up space within the Hub memory, as will every variable (register or Cog memory location) defined by a LONG directive.

RES

Variables may also be defined by using the RES directive. When this occurs, nothing will be placed in Hub memory, for the compiler and assembler the Hub address will remain unchanged, but the Cog address will change as appropriate. Loaded Cog program instructions will therefore have the correct addresses for those variables even though they take up no space in the Hub memory. The consequence of this is that mixing RES definitions with LONG definitions or assembler instructions will cause a mismatch of addressing when the Cog program is loaded to the Cog as a consecutive block of 496 longs and intended execution is unlikely to be achieved.

Variables defined using the RES directive should only appear at the end of an assembler program, after all other LONG definitions and assembler instructions have been specified. RES should only be used for variables which do not require an initial, pre-defined starting value before use, and (to minimise Hub memory use), variables which do not should be defined using RES.

Multiple DAT Sections

Assembly code and shared memory blocks may be split into multiple DAT sections within the same spin file. The address counter for each DAT segment starts where the previous segment stopped (first one starts at 0), unless there is an ORG directive.

References within assembly to DAT variables use the offset within the variable's own DAT section. If this affect is not anticipated, then you could find yourself writing unexpected self-modifying code. The compiler will NOT issue any sort of warning for the following code:

```
DAT
    ORG 0
v1  LONG 0
v2  LONG 0

DAT
    ORG 0
:L  MOV v2,#0     ' This line overwrites the next location in memory w
ith a 0.
    JMP #:L       ' This line of code is corrupted by the time you get
 here.
```

Real programs, step 1: Flashing a LED

The following code will flash a LED connected to the pin 16. For that purpose a small two object files program will be used. The file ledflash.spin should look like:

```
VAR

  long  cog                          ' This is in hub ram

PUB start(pin) : okay

pinmask := |<pin                    ' This can be done because pinmas
k(cog ram) is filled before the data is loaded into
                                    ' the cog, notice pinmask is decl
```

```
ared a long not a res.  After the cog is loaded
                                    ' the value cannot be modified in
 this way.

' Now to start the cog, notice the @entry, look at the assembly, you c
an see where it will begin

okay := cog := cognew(@entry, 0) + 1


PUB stop

'' Stop flashing - frees a cog

  if cog
    cogstop(cog~ -  1)


DAT
                        org                             ' Sets the origin
at zero for this point
entry
                        or      DIRA,pinmask            ' Sets relevant pi
n to an output
                        or      OUTA,pinmask            ' Make pin high
                        mov     time,CNT                ' Current clock lo
aded into time
                        add     time,on_time            ' on_time added to
 time

:loop                   waitcnt time,off_time           ' waits for time t
o pass, off_time added to time automatically
                        xor     OUTA,pinmask            ' toggles pin
                        waitcnt time,on_time            ' waits for time t
o pass, on_time added to time automatically
                        xor     OUTA,pinmask            ' toggles pin
                        jmp     #:loop                  ' loop, the # mean
s that :loop is a literal (something typed into program
                                                        ' rather than a re
gister).

                         ' These variables are in cog ram!

' Initialized data

on_time                 long    80_000*500              ' on time in clock
```

```
 cycles, 500 * 80000 (cycles per millisecond)
off_time              long    80_000*500              ' off time, also 5
00 milliseconds
pinmask               long    0                       ' pinmask can be c
hanged when the program is in HUB RAM

'**** ATTENTION, all res variables MUST come after other variables ***
*********

' Uninitialized data

time                  res     1                       ' res is used only
 to create the label, so no code or other variables
                                                       ' except more res
can be placed here, because no real space is used
```

The file ledflash_demo.spin should look like:

```
{
  Step 1 Flash an LED demo program.
  See ledflash.spin for full details.

}
CON

        _clkmode        = xtal1 + pll16x
        _xinfreq        = 5_000_000

OBJ

  led    : "ledflash"        ' The led object

PUB start

  'start led flashing
  led.start(16)              ' Start the led object (pin 16 flashes a
VGA led on the demoboard)

  repeat   'loop forever more
```

See Also

Assembler Subroutines

Common Assembler Bugs

Atari Joystick Driver & Virtual NES Driver
Written by Jeff Ledger

This file includes a schematic for connecting an Atari joystick to either the Demoboard or Protoboard VGA interface to save I/O for other connections. The interface consists of a DB25male, DB9male, and a few resistors.

The second part of this file is an NES_DRIVER which converts existing code to this interface. Also there is a drop-in replacement for NES code as well.

Books, References and Tutorials

Books

- Game Programming for the Propeller Powered Hydra
- Propeller Manual

References

- Propeller Tricks and Traps - Clever programming tricks, traps to avoid

Tutorials

- Propeller Cookbook - Beginners hardware tutorial
- Programming the Parallax Propeller using Machine Language

Collections

- uController.com - Tutorials section - a collection of tutorials and quick reference sheets

Books People Organization

- bookkeeping services - bookkeeping service by QUALIFIED ACCOUNTANTS removes your anxieties and maintains your accounts at a fraction of what it normally costs
- migration services - Business Sponsorship, Working Holiday Visa & **Permanent Residency** (PR) and Employee Nominated Sponsor (ENS) Visas

Bootloaders

Quick LInks

AiChip Bootloader (AiLoad32 - VB6)
http://forums.parallax.com/forums/default.aspx?f=25&m=211304

Praxis Bootloader (Dot Net 2.0, VS2008)
http://forums.parallax.com/forums/default.aspx?f=25&m=273872

AiChip Bootloader

A working but limited 'proof of concept' bootloader written in VB6 with source code.

More details here

Praxis Bootloader

A front-end for the Parallax Propellent.DLL compile/bootloader combo. Source code for the Dot Net 2.0 Framework, VS2008 available.

More details here

BYTE

A byte is an unsigned integer. Unlike a long which is signed.

BYTE is used as a keyword in 4 different ways:

- In a VAR block
  - `BYTE Symbol`
- In a DAT block
  - `BYTE ...`
- In a method
  - `BYTE [BaseAddressInBytes]`
  - `BYTE [BaseAddressInBytes][OffsetInBytes]`
- In a method
  - `Symbol.BYTE[OffsetInBytes]`

BYTE Symbol

Declaration of a Spin byte variable. When compiling, Spin groups all the byte declarations together in a block after all the long and word declarations, so you can't count on the order of differently sized variables in memory being as in the source. However, all same sized variables will be in the order you declare them.
These variables only exist in Hub memory. They will exist at a place past the binary image created by PropTool.
They are always initialised to zero.
To access them from assembler, you'd have to pass the address of one to the assembly program through the PAR mechanism and use **RDBYTE**/**WRBYTE**.

BYTE ...

Declare a byte aligned label. Size [**BYTE**|**WORD**|**LONG**] indicates how much space to allocate for that labelled location. Size defaults to **BYTE**. Data will be put into the location modulus the Size field. Layout in memory will reflect the order declared in the source, however differently aligned declarations may result in padding.
The data exists in Hub RAM, and may be copied to Cog RAM when starting a Cog. Spin references will use the original in Hub RAM, Assembler references will use the Cog RAM copy (unless done by reference though **PAR** and **RDBYTE**/**WRBYTE**).

BYTE [BaseAddressInBytes]

`BYTE [BaseAddressInBytes] [OffsetInBytes]`

In spin will read/write to a byte in Hub RAM.

`Symbol.BYTE[OffsetInWords]`

In spin will read/write to a byte in Hub RAM. Symbol may be a long, word or byte variable (although as a byte, it'd be more straightforward to use simple array indexing - **`Symbol[Offset]`**).

See also

LONG
WORD
Symbol Address operator

Cog RAM

Cog RAM is the memory local to each Cog. It consists of 496 longs of general purpose RAM for code and data, and 16 special purpose registers. This memory is addressed by longs, $000 through $1EF being general purpose data or instructions, and $1F0 through $1FF being the special purpose registers (see below). Most of the instructions will manipulate the entire 32bits at the given address. The exception is with the code self-modifying instructions MOVS, MOVD, MOVI and the CALL and JMPRET instructions.

- MOVS allows the bottom 9 bits (b8-b0) of a Cog memory location to be modified without affecting other bits.
- MOVD allows the next 9 bits (b17-b9) to be modified.
- MOVI allows the most significant 9 bits (b31-b23) to be modified.
- CALL and JMPRET instructions will also modify the bottom 9 bits (b8-b0) of the RET instruction which relates to the subroutine being called to facilitate a return from Assembler Subroutines.

b31 - b23	b22 - b18	b17 - b9	b8 - b0
Instruction		Destination	Source
MOVI		MOVD	MOVS, CALL, or JMPRET

Special purpose registers:

Address	Name	Type	Description
$1F0	PAR	Read-Only	Boot parameter
$1F1	CNT	Read-Only	System Counter
$1F2	INA	Read-Only	Input states for P31-P0
$1F3	INB	Read-Only	Input states for P63-P32*
$1F4	OUTA	Read/Write	Output States for P31-P0
$1F5	OUTB	Read/Write	Output states for P63-P32*
$1F6	DIRA	Read/Write	Direction States for P31-P0
$1F7	DIRB	Read/Write	Direction States for P63-P32*

$1F8	CTRA	Read/Write	Counter A Control
$1F9	CTRB	Read/Write	Counter B Control
$1FA	FRQA	Read/Write	Counter A Frequency
$1FB	FRQB	Read/Write	Counter B Frequency
$1FC	PHSA	Read/Write	Counter A Phase
$1FD	PHSB	Read/Write	Counter B Phase
$1FE	VCFG	Read/Write	Video Configuration
$1FF	VSCL	Read/Write	Video Scale

*Unimplemented on the Propeller P8X32 chip.

Colors

For the VGA output the number of colors is simply hardwired to 64 colors, (2 bit per base color red, green and blue, six bits in total).
But for composite video the situation is much more complex, and depends for a large part on the software driver.
Not all drivers are available for both PAL and NTSC, and at the moment there are no TV drivers available for the French/Russian SECAM system.

In the standard propeller video configuration (e.g. The Hydra), for NTSC:

- The hydra can produce 16 hues. Each hue can have 4 levels of brightness. = 64 colors.
- Additionally there are 6 monochrome shades from black through white = 6 colors.
- And then there is a set of 16 super saturated colors which overdrive the TV brightness signal and don't play well with the other colors, but can be useful for special effects. = 16 colors.
- Total 86 colors maximum. (135, if all possible byte values are used, however some lower quality signal may result)
- The number of colors you have available in a particular program depends on what palette mode is used by your graphics driver.

Potatohead describes a way of using dithering to produce nearly 400 colors in NTSC.
With custom video hardware, apparently 1.7. million colors are available. But as yet the details are a bit sketchy.
It is safe to say the number of colors possible on the Propeller / HYDRA really depends on the display being generated, amount of RAM added to the on board 32kb, circuit used to interface with the display, and the driver code written.

Notes on High-Color mode

It is useful to know all the byte values, when passed to a WAITVID in high-color mode, that produce colors. There are 126 of these listed in the Color Lookup Table below:

```
$02, $03, $04, $05, $06, $07                    Six intensities
$19, $1a, $1b, $1c, $1d, $1e, $98, $af          15 Hues
$29, $2a, $2b, $2c, $2d, $2e, $a8, $bf
$39, $3a, $3b, $3c, $3d, $3e, $b8, $cf
$49, $4a, $4b, $4c, $4d, $4e, $c8, $df
$59, $5a, $5b, $5c, $5d, $5e, $d8, $ef          High saturation colo
rs, placed
$69, $6a, $6b, $6c, $6d, $6e, $e8, $ff          in table by closest
```

```
hue match.
$79, $7a, $7b, $7c, $7d, $7e, $f8, $0f
$89, $8a, $8b, $8c, $8d, $8e, $08, $1f          Hues are presented i
n this table
$99, $9a, $9b, $9c, $9d, $9e, $18, $2f          vertically, in seque
nce as shown
$a9, $aa, $ab, $ac, $ad, $ae, $28, $3f          in screenie below.
$b9, $ba, $bb, $bc, $bd, $be, $38, $4f
$c9, $ca, $cb, $cc, $cd, $ce, $48, $5f          Start with intensity
 on left, and
$d9, $da, $db, $dc, $dd, $de, $58, $6f          work to the right, s
tarting at
$e9, $ea, $eb, $ec, $ed, $ee, $68, $7f          top of table.
$f9, $fa, $fb, $fc, $fd, $fe, $78, $8f
```

There are 6 additional useful (not high saturation) color values, not mapped into the color table above. (h/t to mpark for catching these!) They are: **$0a, $0b, $0c, $0d, $0e, $0f**
The screen capture below has been updated to show these, and is now the correct phase. This brings the number of non-artifacted, unique colors produced by the Parallax reference video circuit to **135**. These can optionally be mapped into the CLUT above, thus:
09, 0a, 0b, 0c, 0d, 0e, 88, 9f (These form the complete set of input values, some are high saturation and may not display well.)
There are two sets of super saturated colors. On some TVs these appear slightly different, bringing the number of total Propeller colors, generated by the on-board video hardware, to 132 possible input values. As noted above, 86 of these are solid, useful colors. This data provided for completeness where feeding values to a WAITVID is concerned.

Artifacted Colors

The NTSC safe area has an effective resolution of 160 pixels horizontally. (Safe area, shown as lighter grey background in screen shot below.) Pixels clocked (or sized) faster (or smaller) than this, will produce color artifacts in the display. Older 8bit computers and game systems used this technique to obtain more colors than the video hardware was designed for. On the Propeller, it is possible to get a ~400 color display by placing two standard propeller colors together on a display with a pixel clock equal to 320 pixels, for the NTSC safe area. More color combinations, than the ones shown, are possible, using the high saturation color sets in the artifacting process, and the additional colors, not mapped into the CLUT above.

The standard (non artifacted) colors, in the table above, are are in line, at the bottom of the image to the left, and in the strip to the right.

This artifacting technique does deliver 12 grey scales as well, if a monochrome display is desired. For this to work, the color burst signal must not be present.
For resolutions above 160 pixels, stable color can be achieved by placing two pixels together that have the same standard Propeller color. This allows for movement and positioning at resolutions above the 160 pixel limit, without having to worry about heavy artifacting having an impact on the display.
It is important that the left over scan be a multiple of the pixel clock for the above techniques to work.

Color Timing

Simple NTSC displays, similar to those of many older 8bit computers have the 160 pixel limit, and will do artifacting in a stable manner. More complex timing schemes involve interlacing the pixels horizontally, resulting in a 320 pixel NTSC display, or using clever color burst timing for a 224 pixel display.

Reference video output circuits -vs- others

The discussion above surrounds the possibilities presented by the reference Parallax video output circuits, and should be considered standard video output. The nature of the Propeller lends itself easily to other video output schemes as none of the 8 COGs or I/O pins are dedicated to this task in any way. It's a software driven design that makes other video output solutions a matter of code and appropriate output circuits.
An 8 bit R2R ladder can be used to generate a ~160 grey scale display, for example. Because some of the 'colors' actually need to be used for the sync portion of a complete video signal, the actual number of colors, per number of bits applied to the task, will be less than one would otherwise expect from more mainstream graphics systems that perform signal generation in hardware.
Other examples, some in progress, include driving the Propeller video generators in VGA mode to directly control color generation, produce more grey scales, etc...

My Assembler Routine Is Doing Something Weird! What's Wrong?

We've all been there. You've got an assembler routine; in an earlier version it worked perfectly, but now it's doing something really weird. You're read it through over and over, and it looks fine. But still it does something weird. And debugging facilities on the Propeller are non-existent. What on earth is wrong?

Here's a check list for things to do. Do them in order, DON'T skip to the end.

1. Leave it and go do something else. Go make yourself a tea or a coffee. Go for a walk. Sleep on it. It's amazing how often once you stop staring at the screen it occurs to you why it might not be working. Or at least you think of a good way of narrowing down where the defect is.
2. Go through your code checking every single CALL, JMP, JMPRET and DJNZ to make sure that the label that follows has a # in front of it. It's an extremely rare situation when you want a jump of call that isn't immediate. And on the very rare case you do, you should comment it well.
3. Go through your code checking that every single CMP or TEST is followed by WC and/or WZ. The only point of these instructions is to set a flag, but they won't do so unless you explicitly say so. Also, if you are doing signed comparisons, make sure you use CMPS.
4. Go through your code checking every instance of self modifying code. There must always be at least one instruction separating the point at which the instruction is modified and the instruction that has been modified. Rearrange the code or use a NOP to fix.
5. Check that you have not placed a read only special purpose register in the destination field of an instruction unless you really mean it (eg, CMP CNT,reg). The same is true for PHSX (read-modify-write will affect the shadow register only).
6. Check that all your RES lines are at the end of your assembler DAT section. After all LONG, WORD and BYTE declarations.
7. Try the Propeller Assembler Source-code Debugger. It IS possible to single step though your code and examine data as you do so.
8. If you are still stuck, then ask on one of the propeller forums for someone else to take a look. Don't copy more than half a dozen lines into the message itself - the forum software doesn't tend to preserve the formatting at all well. Explain your problem as best you can, and describe what you have already tried, and attach a complete SPIN file or a zip containing the entire project to the post. Anything less is asking for help whilst at the same time ensuring that your helpers will have one hand tied behind their backs. Usually people are very keen to help, but it's unfair to make finding the problem even harder for them than it was for you by not giving them compilable source.

Converting Text Output Display Type

Frequently text output demos of various objects are written using either VGA_text.spin or TV_text.spin, but what if you only have the other type of display? Don't worry, conversion from one type of display to the other is a simple and straight forward process. VGA_text and TV_text are symmetric objects, meaning the same methods (with the same argument types) are located in both objects. Converting the demo is a 3 step process:

1. In the the OBJ section, change "VGA_text" to "TV_text" (or "TV_text" to "VGA_text")
2. Change the basepin argument in the start method for the text display. For the Propeller Demo Board the basepin for VGA is 16 and the basepin for TV is 12. For basepin values of other Propeller development boards, check this page or the board's documentation. The lowest pin connected to the output is the basepin value.
3. Color values for VGA and TV are different. VGA color values are 2 bits for each primary (for a total of 6 bits) with bits 0 & 1 containing the Blue component, bits 2 & 3 containing the Green component and bits 4 & 5 containing the Red component. Online color picking utilities can assist in the selection of VGA color values. TV color values are composed from chroma and luminence values, bits 0-2 is the luminance value (greyscale brightness, 2=black and 7=white *), bit 3 indicates the chroma value is used (this bit should be set to 1 for non-greyscale colors), bits 4-7 is the chroma value (which angle of the color wheel the color lies on). Graphics_Palette.spin located in Examples\Library of the Propeller Tool distribution provides an interactive means for picking TV color values. This third step is only needed if the color apperance is important for the demo.

After these three steps are taken the demo has been converted to the other display type.

*) 0 is SYNC-Level and 1 is "superblack" which should be avoided. When you use color (Bit 3 set) it is also recommended to avoid level 7, as the overlaid color signal cannot fit on top.

I've Found some Propeller Code, now what?

As annoying as the idea is, the reality is all computer code comes with both a copyright and a license. For those not having any real experience with this, Here's a summary of a recent forum thread on this topic, that should leave you with a fairly good idea of how things work and what you need to do, if you want to use / incorporate other code into your own work.

Copyright

The author of a creative work automatically has copyright, it happens at the moment of creation. Copyright is essentially having a say over what is permissible where derivative works, or distribution is concerned. Derivatives are works that embody the authors work, in whole, or in part, to form a new work. Distribution is essentially duplication of said work, from one party to another. Use of the work, almost always involves both of these things. (one copies code from storage to memory, in order to execute, for example.)

License

Is essentially the terms of use, as dictated by the author. **Best form is this: One does not use software one didn't write, without a license**. That's a really easy way to sort this stuff out. Everything has a license of some kind! (I, as the initial author of this Wiki page, never knew this early on!) All anybody needs to do is be able to cite the license they are operating under, or put another way, that grants their entitlement to the work in question. Easiest way is to give credit, and cite the source for the license there.

In a nutshell, it's best to do some work to figure out what the license on found code is, or just contact the author for clarification and potentially permission. Having done this, good form means giving credit to the original author, and a reference to the license granting your use, or a statement to the effect of: "Used with Permission". Others then, seeing your completed work, understand what is yours, what isn't, and how you came to be in a position where the building and distributing of it is authorized.

Distribution really is the key element in all of this. If one does things on their own machines, for their own personal purposes, this stuff can be largely ignored. Once one moves beyond that, it's important to work through this stuff, if things are not clear.

Code found in the Parallax Object Exchange, comes with the MIT license. In this case, it's really easy! Just credit the author, and cite the object exchange, and you are good to go. Do what you will, but do also consider putting something into the pool for others to enjoy in like kind. The Object Exchange used to have a "Public Domain" like license scheme. For a short while, you may well find code not yet MIT licensed. Read the program headers, and or README, or LICENSE files to know what the terms are.

If you have already obtained code from the Object Exchange, and it was licensed under the old terms, that license still applies. This change is really about preserving copyright for the owners, and giving credit

where credit is due, while at the same time permitting a wide range of permitted uses.

eg: Magic Bubble code V 2.1 by Bob the Coder, obtained and used under the MIT license, Parallax Object Exchange, 2008

Code found elsewhere is more grey. **Best practice is for authors to state their license terms, either directly, or by reference to a known and established license, in the program header, so that others may know the default terms of use.**

Sometimes you will see this, other times you won't. You also will see the license in a separate file, with a name like README, GPL, license, etc... It just varies, with some people not caring enough to worry about any of it, to others fairly concerned about commercial use. If you can't easily sort it out in a few minutes, it is time well spent with a brief contact with the author to get permission or maybe just clarify the license intent.

Such a contact can be really simple. Use a forum PM, e-mail, IM, or something and just let the author know what you are wanting to do. "Hi, this is Joe and I'm working on the Propeller. Is it ok if I release a modified version of your [whatever] code that works with my kind of development board / set-up?"

Once that exchange has happened, then it's just a matter of a credit line, somewhere in your program header, thus:

Super sensor driver V1.x modified with permission from somebody@somedomain.net (their real name maybe), 2008
Keyboard driver V3.0, BSD licensed, contributed by somebody@somedomain.com, 2007

You may find code on an authors web page. If this is the case, generally they will state their terms there, or provide contact info. If the code is in a repository of some kind, code bodies are typically categorized by their functionality and license type. You can consult the repository FAQ, and generally read the licenses directly, as they are normally established ones, such as GPL, BSD, Creative Commons, etc...

At this time, the most common cases for finding code are here in the Wiki, Parallax Forum postings (ask authors about those --everybody is cool about it), and the Parallax Object Exchange.

Generally speaking, most people who contribute code on-line have some understanding of how complex or confusing these matters can be. Contacting them is perfectly OK, and welcome! Worst case, you gain an understanding of what your options are and can make solid choices from there. Best case, you make a friend or two! It may seem problematic to track these things down, and generally, that perception is reality. All that can be said on the matter is it gets easier after the first time.

Cracking Open the Propeller Chip

Decoding the Spin Interpreter

Chip Gracey (Parallax) : Can anyone out there figure out how to get the proper binary image of the current Spin interpreter from ROM?

Harley Shanko : Did I just hear a challenge?

Chip : Yep! Maybe it hasn't been pursued, out of consideration to Parallax, but if someone posts the correct binary, I'll post the original source code.

Original thread : here

Overview

The Spin Interpreter along with the bootloader is stored in ROM, according to the Propeller Memory Map between byte addresses $F002 and $FFFF. Although this ROM area can be easily read by a Spin or PASM program (as other data in ROM can be, character bitmaps and trigonomic tables), the data returned is encrypted. This data is decrypted by on-chip and unknown hardware as it is loaded into a Cog for execution. As Chip has written ...

"The booter is at $F800 and the interpreter is at $F004. You will not be able to disassemble these programs, though, since the data is scrambled and only gets unscrambled by the HUB during launching. This is the only 'code protection' that the chip has and it's designed to slow down others from making me-too Propeller-like chip products".

Original thread : here

Encryption Mechanism

The encryption method used is not known but the Propeller chip does contain an LFSR which is used during program download and it very possible that an LFSR is used as apart of the decryption process. The LFSR taps used are documented.

A reversible LFSR is also used to randomise Spin variables. Although this is most likely implemented as code within the interpreter itself (there are no LFSR related Cog opcodes) there may be similarities between that and any LFSR used for ROM decryption. The randomisation algorithm can be found here

Chip has referred to the ROM image as "scrambled" rather than "encrypted" so the mechanism used could be quite simple, even if not easy to determine. Obvious "scrambling" mechanisms would be simple XORing, bit rotation, bit reversal and address bit re-ordering (PASM code stored non-sequentially). Any

number of these methods, plus LFSR could be applied together, along with decryption being accumulative, the previously decoded value affecting the decoding of the next.

The encyption could be byte, word or long oriented. While loading Cog is usually talked of in terms of loading 512 longs, it could equally be the loading of 2K bytes.

Approaches to Decoding the Spin Interpreter

Although the Spin Interpreter reads Ram to initally configure the start address and stack, then reads Ram to interpret the bytecode, the program counter, stack pointer and related registers are all held within the Cog. There appear to be no Spin bytecodes which can extract data from within the Cog nor be subverted to do so. Trying to create a 'buffer overflow' type of attack to leave code in a Cog which could dump its contents out does not look possible.

The Spin Interpreter is launched by a CogInit of the PASM code held at $F004 with the PAR register pointing to a 12-byte block of memory ($0004 at first Spin Interpreter boot-up) which defines where the Spin bytecode to execute and stack is. By executing a CogInit of the assembler code above $F800 it may be possible to load a PASM program held low in Hub Ram which would be loaded as part of the 496 long load into Cog (through address wrap-round) leaving that PASM program plus some decrypted ROM in the Cog, and with a fair wind, see that PASM code executed and reveal the decrypted bytecode. Although the decrypted PASM code is not the actual Spin Interpreter (it's the bootloader) it would help in verifying any decryption algorithm determined. Whether such PASM code (at the end of Cog memory) would ever get executed is debatable and the odds would seem to be against it.

The best approach would appear to be to dump the ROM between $F000 and $FFFF and attempt to decode that either on chip or off-chip using a PC.

Spin Interpreter ROM Image

An extracted Spin Interpreter ROM Image (in its encrypted form) is available here as SPIN.ZIP (in both a .BIN and a .HEX form) from
Peter Jakacki and a program to extract a ROM image is provided by dartof.

Die Images

A - COG RAM - (view of the ISDR Word Line drivers)
B - HUB RAM - (view of the Column Address Decoder)
C - HUB ROM - (view of the 4-16 Decoder driver)

Here

Possible Helpers

The Spin Interpreter needs to know where the information is from which it initialises its program counter and stack pointer. This is provided by way of the PAR register set through a CogInit and thus the Interpreter should use the PAR as a source register fairly early on in its execution.

It would not be surprising to find an early 'rdbyte', 'rdword' or 'rdlong' using PAR and, as PAR is read only, a 'mov' from PAR and subsequent 'rdbyte', 'rdword' or 'rdlong' using the register PAR is moved to. It would not be unreasonable to expect the first Cog instruction to be a 'mov'. Concentrating on just the early part of the interpreter code could make a brute force attack on the encryption more feasible.

Given that there would seem to be only a limited amount of run-time data storage used within the run-time interpreter, it would seem unlikely that the initialisation code would be within an area later in the code which would subsequently be re-used for data storage. If that were the case, it would seem likely that a 'jmp' or 'jmpret' would appear early in the code.

Cracking Open the Propeller Chip

Decoding the Spin Interpreter

Chip Gracey (Parallax) : Can anyone out there figure out how to get the proper binary image of the current Spin interpreter from ROM?

Harley Shanko : Did I just hear a challenge?

Chip : Yep! Maybe it hasn't been pursued, out of consideration to Parallax, but if someone posts the correct binary, I'll post the original source code.

Original thread : here

Overview

The Spin Interpreter along with the bootloader is stored in ROM, according to the Propeller Memory Map between byte addresses $F002 and $FFFF. Although this ROM area can be easily read by a Spin or PASM program (as other data in ROM can be, character bitmaps and trigonomic tables), the data returned is encrypted. This data is decrypted by on-chip and unknown hardware as it is loaded into a Cog for execution. As Chip has written ...

"The booter is at $F800 and the interpreter is at $F004. You will not be able to disassemble these programs, though, since the data is scrambled and only gets unscrambled by the HUB during launching. This is the only 'code protection' that the chip has and it's designed to slow down others from making me-too Propeller-like chip products".

Original thread : here

Cracked Open

After a flurry of activity the Propeller was cracked open and the interpreter revealed. True to his word Chip released the original source code of the Interpreter and Bootloader.

The 'cracking' process as it evolved and interpreter source can be found here

The key to reverse engineering turned out to be embraced in Chip's description that the data was "scrambled" rather than "encrypted". The mechanism used was simple bit swapping of data. Although simple, it had been good enough to kept people away from attempting to decode the data or, if they tried, from doing so. Had Chip not thrown down the challenge, and suggested it was perhaps more possible than people were thinking it would be, then it may have remained unencoded.

Even knowing the mechanism used it wasn't a simple and straight forward process to decode the entire interpreter. Various approaches were taken, brute force, 'good guessing' and statistical analysis.

Further Developments

The full source code for INTERPRETER, BOOTER, and RUNNER was released by Chip here

Original Copy of this Page

This page has been updated to reflect the release of the interprter source code, the original page discussing possible approaches and useful help can be found at the link here : Original Page

Data Storage

- Example

Debuggers and Emulators

Spin Debuggers

View Port

Commercial tool now adopted by Parallax. Includes a spin debugger, which is only one currently available.
Windows based.

More details here

SPUD

SPUD stands for Spin PASM Unit Debugger.

Spud is a spin debugger that is similar to gdb, and is available for Linux and Windows running Cygwin. Released as open source in its forum thread here.

Cluso SPIN Debugger

Open Source Debugger capable of single stepping SPIN code. Uses a serial interface to allow stepping and display of data, but is involved to setup your code to test. Does not have a source display. Current release is 275/276 from its forum thread here
Also there is a similar version forP ASM.

Asm Debuggers

BMA PASM Debugger

Here is a simple on chip PASM debugger. The BMAdebugger does not use a GUI.
Debugging is most effective when used in conjunction with BSTC compiler .list files.

- Small and simple on chip PASM debugger
- No Windows or other PC program required

- Feature rich command set
- Multi-COG debug ability

from its forum thread.

KISS Debugger

Commercial tool that has on chip assembly level debugging. Standalone, using propeller based IO, no need for PC at all.
More details here

PASD

The debugger system consists of a PC program, a spin-object and a short debug the kernel, at the beginning of the debug code to be inserted.

The debug kernel is only 12 longs large, and allows you to communicate with the PASD spin driver, in a separate Cog runs. This spin-driver communicates via the serial programming interface with the PC and to display current program. Apart from the pins 30 and 31 remain, all IOs of the propeller available during debugging.

More details PASD

POD Debugger

PASM Debugger. Supports breakpoints and single stepping PASM code running on the target hardware.

POD is propeller source that you build into your project, and add a few hooks to the PASM to be debugged.

Can display and single Latest version can be controlled via PropTerminal or similar. The original versions used a keyboard and TV monitor directly attached to the Propeller for input and display.

More info and releases are in this thread.

Cluso PASM Debugger

Open Source Debugger capable of single stepping PASM code. Uses a serial interface to allow stepping and display of data, but is involved to setup your code to test. Does not have a source display. Current release is 275/276 from its forum thread here
Also there is a similar version for SPIN.

Monitors

A monitor is a simple mechanism for observing debug output, when a debugger is not available. There are a number of spin objects that can be used for this use, that use serial, video or LCD to display debug information.

Monitor.spin dumps the main memory contents using a serial interface, with some support for iteration with the terminal.
SerialMirror.spin can be used to dump out diagnostic info from spin code.

Simulators

pPropellerSim

pPropellerSim is a powerful full-featured Propeller Simulator/Assembler and Debugger. A built-in editor with syntax highlight allows you to program in propeller assembler, to compile and to simulate your code in a fully working COG (no hw at this time). A high compatibility level with Parallax' PropellerTool allows for easy and rapid transfer of DAT sections.

It is program in the Java programming language which makes it able to run on Solaris, MacOS X, Linux and more.

The manual on-line can be found here and it can be downloaded from here

GEAR

GEAR is a open source C# program the emulates the inner workings of the Propeller chip. It is able to emulate multiple spin and PASM cogs concurrently. GEAR also has a plug-ins interface that allows external hardware to be emulated. Current plug-ins include a VGA display, and a pin toggling (stimulus)

module.

Not in active development since 2007.

The latest version of GEAR can be found at the project page or on the forums.

PropList

PropList from AiChip Industries is a Windows Command Line utility to disassemble Spin Bytecode and PASM instructions from a .binary or .eeprom file creating a symbolic list file (.lst) or ByteCode Assembler source (.asm) for further processing.

The utility traverses the code image for objects, methods and PASM code to maximise the amount of usable disassembly that can be determined. The program is written in PowerBasic (PB/CC) with source code included.

More details : here

Development Board Differences

When programming for the Propeller it's useful to know the output pin and crystal differences between the various development boards.

If you have the details for another board please add it. If the pins are identical to one of the dev boards already listed, then don't use another column, just add the name of the board in the heading of the existing column.

	Hydra	Hybrid	**Protoboard**	Demoboard	SpinStudio	Boss Board	Propeller Prof. Dev. Board	Chameleon
XTAL	10 MHz	6 MHz	5MHz	5MHz	5MHz	G11 - G13	5 Mhz (re movable)	5 MHz
P0	Debug LED	Debug LED	Available	Available	Socket A Pin 0	M3	Available	Expansion Port (0)
P1	NET_RX_CLK	Expansion Port (10)	Available	Available	Socket A Pin 1	M4	Available	Expansion Port (1)
P2	NET_TX_CLK	Expansion Port (9)	Available	Available	Socket A Pin 2	M5	Available	Expansion Port (2)
P3	Joystick CLOCK	Joystick CLOCK	Available	Available	Socket A Pin 3	M6	Available	Expansion Port (3)
P4	Joystick Latch	Joystick Latch	Available	Available	Socket A Pin 4	M7	Available	Expansion Port (4)
P5	Joystick 0 Data	Joystick 0 Data	Available	Available	Socket A Pin 5	M8	Available	Expansion Port (5)
P6	Joystick 1 Data	Joystick 1 Data	Available	Available	Socket A Pin 6	M9	Available	Expansion Port (6)
P7	Audio Out	Audio Out	Available	Available	Socket A Pin 7	M10	Available	Expansion Port (7)
P8	Mouse	SD CARD d0	Available	Microphone	Socket B Pin 0	M15	Available	SPI SS
P9	Mouse	SD CARD clk	Available	Microphone	Socket B Pin 1	M16	Available	SPI MOSI

P10	Mouse	SD CARD di	Available	Stereo Audio Out	Socket B Pin 2	M17	Available	SPI MISO
P11	Mouse	SD CARD cs	Available	Stereo Audio Out	Socket B Pin 3	M18	Available	SPI SCLK
P12	Keyboard Data Out	Keyboard Data I/O	Available	Video Out	Socket B Pin 4	M19	Available	Video Out LSB
P13	Keyboard Data In	Keyboard Clock I/O	Available	Video Out	Socket B Pin 5	M20	Available	Video Out
P14	Keyboard Clk Out	Mouse Data I/O	Available	Video Out	Socket B Pin 6	M21	Available	Video Out
P15	Keyboard Clk In	Mouse Clock I/O	Available	Video Out	Socket B Pin 7	M22	Available	Video Out MSB
P16	VGA Out	Expansion Port (1)	*VGA Out	VGA Out	Socket C Pin 0	GHJ22	Available	VGA Out
P17	VGA Out	Expansion Port (2)	*VGA Out	VGA Out	Socket C Pin 1	GHJ21	Available	VGA Out
P18	VGA Out	Expansion Port (3)	*VGA Out	VGA Out	Socket C Pin 2	GHJ20	Available	VGA Out
P19	VGA Out	Expansion Port (4)	*VGA Out	VGA Out	Socket C Pin 3	GHJ19	Available	VGA Out
P20	VGA Out	Expansion Port (5)	*VGA Out	VGA Out	Socket C Pin 4	GHJ18	Available	VGA Out
P21	VGA Out	Expansion Port (6)	*VGA Out	VGA Out	Socket C Pin 5	GHJ17	Available	VGA Out
P22	VGA Out	Expansion Port (7)	*VGA Out	VGA Out	Socket C Pin 6	GHJ16	Available	VGA Out
P23	VGA Out	Expansion Port (8)	*VGA Out	VGA Out	Socket C Pin 7	GHJ15	Available	VGA Out
P24	Video Out LSB	Video Out LSB	*Mouse Data I/O	Mouse Data I/O	Socket D Pin 0	GHJ10	Available	Audio Out
P25	Video Out	Video Out	*Mouse	Mouse	Socket D	GHJ9	Available	Status

			Clock I/O	Clock I/O	Pin 1			LED
P26	Video Out	Video Out	*Keyboard Data I/O	Keyboard Data I/O	Socket D Pin 2	GHJ8	Available	Keyboard Data
P27	Video Out MSB	Video Out MSB	*Keyboard Clock I/O	Keyboard Clock I/O	Socket D Pin 3	GHJ7	Available	Keyboard Clk
	Hydra	Hybrid	**Protoboard**	Demoboard	SpinStudio	Boss Board	Propeller Prof. Dev. Board	

Same for all boards:

P28	EEPROM Clk
P29	EEPROM Data
P30	Serial RX
P31	Serial TX

*Pins 16-27 are available until the "Accessory Kit" is added to the Protoboard.

Tools to help you in developing software for the Propeller chip.

- Propeller Tool - Enhancement Requests
- Debuggers, Emulators and Disassmblers
- Bootloaders
- Code Formatter - for forum postings
- Developing in GNU/Linux Environments
- Homespun Spin Compiler
- Programming in C - Catalina
- Propeller Assembler Source-code Debugger
- BST, Brads Spin Tool: Mac, Linux and Windows native development
- Sphinx and SphinxOS: Propeller-based Spin compiler and SphinxOS
- LAS - Largos Assembler, supports standard PASM and LMM with assembler extensions for easy LMM programming

The DK Graphics Driver was written specifically for the game Dodgy Kong. However there is no reason why it may not be extracted and used to power other games.

Specs

Screen Resolution (WxH)	256x224
Tile Resolution	8x8
Tiles	32x28
Tile Palette	4 colors per individual tile
Sprite Resolution	16x16
Sprite Palette	4 colors per individual sprite
Number of sprites	34
Sprites per line	Depends on number of cogs used
Cogs needed	3+ More sprites or more tile palettes require more cogs
Files	sw_dk_gfx_renderer_015.spin sw_dk_tv_drv_022.spin

How it works

DK uses 3 or more cogs working together. 2 or more rendering cogs and a single TV driver cog. The rendering cogs work in parallel creating data for a single scan line at a time. The TV cog takes this data and sends it to the propeller graphics hardware.

DMX

DMX512-A is an RS-485 based communications protocol that is most commonly used to control stage lighting and effects. Wikipedia has lots of background information. And there is a web site dedicated to DMX, and another that has the full standard

Drivers

On the Object exchange there is a DMX-512A Receive Driver created by Timothy D. Swieter

Jon Williams' November, '09 *Spin Zone* column (#3) discusses DMX. You can download it from Parallax.

There is also a thread containing a DMX transmitter by Teva McMillan

Projects

Timothy D. Swieter created a DMX controlled light show built into his bookshelves and there's a nice video of it on YouTube.

Download Protocol

The following is based on the booter.spin file released by Chip Gracey of Parallax as an attachment to his message posting to the Parallax Propeller forum titled: "Propeller ROM source code HERE".

Though there is no explicit copyright in the message or the attached source code, this Wiki editor believes U.S. copyright law interprets that as copyrighted with all rights reserved. However, this Wiki editor is not a lawyer.

In the Parallax Propeller forum thread titled "Standalone, cross-platform Propeller assembler", Chip Gracey of Parallax also posted a message with the attachments: source for a download program written in Delphi Pascal, and a text explanation of the download protocol.

Also posted by Chip Gracey of Parallax is a SPIN object for loading a Propeller from a Propeller, in the thread titled "Propeller Loader"

Chip Gracey's "Minimal Spin bootstrap code for assembly language launch" is also a handy reference.

Three Bit Protocol (3BP)

The Propeller uses an adaptive Three Bit Protocol (3BP) for serial signaling which works at most any bit rate, at the cost of reduced throughput because of signaling overhead. 3BP symbols begin high with an idle line, followed by either one or two bit times low to signal a one or zero bit respectively. These two bit times are referred to as T1 and T2.

As part of the handshake sequence, the Propeller calculates a threshold of 1.5 bit times in system clock cycles. If the RXD pin is low for longer than this threshold, a zero symbol is received, otherwise a one symbol is received.

Since the Propeller boots using its internal RC clock (RCFAST), which can vary in frequency from 8 MHz to 20 MHz, all timings are at best approximations. Any timeouts are calculated assuming a 20 MHz clock.

3BP Meets 8N1

Assume that 3BP(X) and 8N1(X) map data X into a unique symbol that can be signaled on a serial line, then:

- **3BP Symbols**
- 3BP(idle): 1

- 3BP(0): Two bit times low which is 001 transmitted least significant bit first.
- 3BP(1): One bit time low, which is 101 transmitted least significant bit first.

- **8N1 Symbols**
- 8N1(idle): 1
- 8N1(8 data bits): start bit (0) + 8 data bits, least significant first + stop bit (1)

Transmitting one 3BP symbol per 8N1 symbol is simplest, but very inefficient:

- 3BP(0) = 8N1(FE)
- 3BP(1) = 8N1(FF)

It is possible to pack one or more 3BP symbols into a traditional ten bit time 8N1 serial symbol!

Fixed length 3BP-to-8N1 symbol packing looks like this:

8N1 Symbol	Line State	3BP Symbol
Idle or Stop	1	Idle
Start	0	T1
Bit 0	Data Bit 0	T2 or Idle
Bit 1	1	Idle
Bit 2	0	T1
Bit 3	Data Bit 1	T2 or Idle
Bit 4	1	Idle
Bit 5	0	T1
Bit 6	Data Bit 2	T2 or Idle
Bit 7	1	Idle
Stop	1	Idle

It is also possible to do variable length packing to reduce the back-to-back idle bit times, but it is more complex to do as it depends on the symbols being sent.

Handshake Sequence

All communication with the Propeller is done using 3BP.

There is a timeout on completion of the entire handshake sequence of 375,000 cycles (150 ms at 20 MHz). There is a reset delay of 50 ms, and then COG 0 is loaded, which takes 512 longs * 16 cycles per hub access = 8,192 cycles.

Important: With a 150 ms limit for the handshake, which requires 251 8N1 symbols to be transmitted (2510 bits), the closest standard minimal bitrate required is 19200 BPS. With a more conservative 100 ms for the handshake, 38400 BPS is recommended as a minimum bitrate. For lower bit-rates, multiple 3BP symbols must be sent per 8N1 symbol.

1. Transmit on RXD: 3BP(idle).
2. Reset the Propeller (on Parallax development boards: set DTR low, wait at least 10 ms, set DTR high).
3. Wait 100 ms. This leaves roughly 100 ms to complete the handshake.
4. Transmit on RXD: 3BP(0) followed by 3BP(1), which is equivalent to 8N1(F9). This sets the bit timing threshold.
5. Transmit on RXD: 250 bits from a Linear Feedback Shift Register (LFSR):

```
Initialize the LFSR to ASCII 'P' (50 hex)
    Repeat 250 times:
        Transmit on RXD:  3BP(LFSR bit 0), which is equivalent to 8N1(
FE) if zero or 8N1(FF) if one.
        Set rotate-in bit to even parity of LFSR AND B2.
        Rotate LFSR left one bit.
```

which is equivalent to:

```
Initialize the LFSR to ASCII 'P' (50 hex)
    Repeat 250 times:
        Transmit on RXD:  3BP(LFSR bit 0), which is equivalent to 8N1(
FE) if zero or 8N1(FF) if one.
        Set LFSR to (shift LFSR left 1 bit) OR (
            (
                    (shift LFSR right 7 bits)
                XOR (shift LFSR right 5 bits)
                XOR (shift LFSR right 4 bits)
                XOR (shift LFSR right 1 bit)
```

```
            ) AND 1
        )
```

The first 250 bits of the LFSR encoded with 8N1 are:

```
FE FF FE FF FF FF FE FE FF FF FF FF FE FF FE FF
FF FF FF FF FE FE FE FF FF FF FE FE FF FE FF FE
FE FE FF FF FF FF FE FE FE FE FF FE FE FF FE FE
FF FE FF FF FF FF FE FE FF FE FE FE FF FE FE FE
FF FF FE FF FF FE FF FF FF FE FE FE FF FE FF FF
FE FE FF FF FE FE FF FE FF FF FE FF FF FE FE FF
FE FE FF FF FF FE FF FE FF FE FF FE FF FF FF FE
FF FF FE FF FE FF FF FE FF FE FE FF FF FF FF FF
FF FF FF FE FE FF FF FE FF FF FF FF FE FF FF FF
FE FF FE FE FE FE FE FE FE FF FE FF FE FF FF FE
FE FE FF FF FE FE FF FF FF FE FE FE FE FE FE FF
FF FF FF FF FE FF FE FE FF FE FF FE FF FE FE FF
FE FE FE FE FE FF FE FE FE FE FF FF FF FE FF FF
FF FF FF FF FE FF FF FE FE FE FE FF FF FE FE FE
FF FE FE FF FF FE FE FE FE FE FF FF FE FF FE FE
FE FF FE FF FE FE FF FF FE FF
```

Note that the LFSR only requires an 8 bit wide register to calculate, as all higher bits are ignored.

Identify Sequence

All communication with the Propeller is done using 3BP.

The Propeller relies on received pacing symbols to frame a 3BP response for the next 250 bits of the LFSR.

There is a timeout on transmitting each bit of 250,000 cycles (100 ms at 20 MHz), and it is reset with each bit the Propeller sends.

1. Receive on TXD: the next 250 bits of the LFSR, and verify.

```
Repeat 250 times:
    Transmit on RXD:  3BP(1) followed by 3BP(0) for pacing, which is e
quivalent to 8N1(F9).
    Receive on TXD:  3BP(0) or 3BP(1) for the next bit of the LFSR, an
```

```
d verify
```

The next 250 bits of the LFSR encoded with 8N1 are:

```
FE FF FE FE FE FE FF FE FF FF FF FE FE FF FF FF
FF FE FF FE FF FF FF FF FF FE FE FE FF FF FF FE
FE FF FE FF FE FE FE FF FF FF FF FE FE FE FE FF
FE FE FF FE FE FF FE FF FF FF FF FE FE FF FE FE
FE FF FE FE FE FF FF FE FF FF FE FF FF FF FE FE
FE FF FE FF FF FE FE FF FF FE FE FF FE FF FF FE
FF FF FE FE FF FE FE FF FF FF FE FF FE FF FE FF
FE FF FF FF FE FF FF FE FF FE FF FF FE FF FE FE
FF FF FF FF FF FF FF FF FE FE FF FF FE FF FF FF
FF FE FF FF FF FE FF FE FE FE FE FE FE FE FF FE
FF FE FF FF FE FE FE FF FF FE FE FF FF FF FE FE
FE FE FE FE FF FF FF FF FF FE FF FE FE FF FE FF
FE FF FE FE FF FE FE FE FE FE FF FE FE FE FE FF
FF FF FE FF FF FF FF FF FF FE FF FF FE FE FE FE
FF FF FE FE FE FF FE FE FF FF FE FE FE FE FE FF
FF FE FF FE FE FE FF FE FF FE
```

2. Receive on TXD: the 8 bit Propeller chip version.

```
Repeat 8 times:
    Transmit on RXD:  3BP(1) followed by 3BP(0) for pacing, which is e
quivalent to 8N1(F9).
    Receive on TXD:  3BP(0) or 3BP(1) for the next bit of the Propelle
r chip version.
```

Transfer Sequence

All communication with the Propeller is done using 3BP.

All data is sent as 32 bit longs, least significant bit first. There is roughly 100 ms timeout on receiving each long.

1. Transmit on RXD a 32 bit command:
   Command = 0: Shutdown.
   Command = 1: Load RAM and launch.
   Command = 2: Load RAM, write and verify EEPROM, and shutdown.

Command = 3: Load RAM, write and verify EEPROM, and launch.
2. Transmit on RXD; 3BP(N), a 32 bit count of the number of 32 bit longs to follow. There is a maximum of 8192 longs allowed (32 KB).
3. Transmit on RXD: 3BP(N x 32 bit longs). If less then 8192 longs, then the remaining 8192 - N are set to 0. The 8192 longs must have a 0 checksum.
4. Transmit on RXD every 10 ms: 3BP(0) followed by 3BP(1) for pacing, which is equivalent to 8N1(F9).
5. Receive on TXD: the checksum result 3BP(0) = passed or 3BP(1) = failed, which is equivalent to 8N1(FE) = passed and 8N1(FF) = failed.

DeSilva has prepared a quite readable article about self-clocking data streams and the "Three-Bit-Protocol" here (Propeller Forum) some time ago...

Editing the Propeller Wiki

Who can edit this Wiki?

Anyone can. If you see something that is wrong, or needs a extra bit of explanation, or is missing some technical detail, or could just do with rearranging, just dive in and change it. Be bold!
You don't even have to create an account to do the editing, although if you do, you'll be better able to keep track of what edits you've done.
If you want to create a whole new page, you will have to create an account though.

Can I create a page about my project or product?

Sure you can. If it will be of interest to other Propeller users, it's welcome here. And who better to write about it than you? You might already have some content that you've written for the manual or your website that you want to reproduce here. But think factual and informative rather than plugging and advertisement.

Style

Think of the Wiki as an encyclopedia of topics related to the Propeller. Try to keep it impersonal. If you use the personal pronoun "I", that might make sense at the time you are first creating a page, but over a persiod of time, a page will have text written by multiple people, and "I" won't make any sense any more. Don't worry about being comprehensive. If you omit things and they're important, someone else can add them later.

Editing

Just hit the "Edit This Page" button at the top of the page you want to change. You start with a Visual Editor, but you can change to a Text Editor which accepts standard wikitext markup if you prefer. If ever you get strange things happening in the Visual Editor, it's often a good idea to switch to the Text Editor to see what's wrong.

Creating a new page

Click "New Page" in the list of actions at the top of the column on the left. You'll have to be logged in to see it. You then get to choose the page name. Try to make this brief but descriptive of the content. It appears in lists elsewhere in the system and needs to be obvious.

Formatting

Title

At the top of every new page, place a title. Often this is the same as the page name, but here you can be more verbose. For example the name of this page is "Editing the Wiki", but the title is "Editing the Propeller Wiki". Format this as **Heading 1**, either using the drop down menu or by placing **=** before and after the title.

Section Title

Then for each section of the page, give a section title. Format this as **Heading 2** (use == before and after the title) and subsections as **Heading 3** which is marked with ===

Code

For multiline snippets of code, create a code section with place **[[code]]** at the start and end. For the odd keyword that you want to display in-line, make the font monospaced bold, by putting **{{**** at the start and ****}}** at the end.

Example

PUB Main

Fast-Track for the PropJavelin

This document is a fast-track guide to getting the PropJavelin up and running for those who have no experience with Parallax's Javelin Stamp Product.

PropJavelin is a project to implement the functionality of the Javelin Stamp on the Propeller Chip. This is the implementation of a JVM which runs on the Propeller to allow Java(TM) programming of the Propeller. Java program development is undertaken using a modified version of the Javelin Stamp IDE.

This document assumes the user does have experience of Windows XP, installing applications under Windows XP, has the Parallax Propeller Tool installed and working and is familiar with its use.

It is not however necessary to be familiar with the Javelin Stamp, its IDE or the Java programming language to get a PropJavelin running, although an understanding of Java will be necessary to program and fully utilise the PropJavelin.

Please feel free to update this document to aid other users who are unfamiliar with the Javelin Stamp and PropJavelin.

Install the Parallax Javelin Stamp IDE

From the Parallax website, Downloads for the Javelin Stamp

Download Javelin Stamp IDE Installer - Version 2.0.3 (Win2k, XP & Vista) 2.2MB (EXE)

You may also wish to download the Javelin Stamp Users Manual Version 1.0a 1.4MB (PDF)

Execute the downloaded JvlnSIDEsetupv203.exe to install the Javelin Stamp IDE.

Launch the Javelin Stamp IDE and click Help/About; this should show "Version 2.0.3 Build 0". Click on the pop-up to close it. Close the application.

Get the PropJavelin Updates

From page 8 of the thread "JVM for Prop", download the three files ...

1. javelin.zip 763KB (ZIP)
2. jvm.zip 48KB (ZIP)
3. TerminalTest2.java 2KB (TXT)

You may wish to check for later versions of these files. The rest of this document will assume that the above files were downloaded.

Install the PropJavelin IDE

Unzip the downloaded javelin.zip and extract javelin.exe to the directory where the Javelin Stamp IDE was installed to. This is likely to be "C:\Program Files\Parallax Inc\Javelin Stamp IDE" on many systems.

Launch Javelin.exe and click Help/About; this should show "Version 2.2.0 Build 4". Click on the pop-up to close it. Close the application.

Install the PropJavelin Firmware on the Propeller

Create a directory to hold the PropJavelin firmware (the JVM) and extract the entire contents of jvm.zip to it.

Launch the Propeller Tool and load the jvmMain.spin file. Use "F9" to syntax check the software and confirm all required files have been properly installed. This may take a few seconds on a slower PC.

Modify the _CLKMODE and _XINFREQ constants at the top of jvmMain.spin to match the Propeller hardware you will be using. Note that the default is for the Spin Stamp product which uses a 10MHz crystal and will likely have to be changed if using one of the other Propeller development platforms such as the ProtoBoard.

If not already configured, set the commPort constant to 1.

Connect your Propeller hardware and Use "F7" to identify the COM port which will be used by the Propeller Tool for downloading and remember it - this information will needed during the next step.

Use "F11" to burn the PropJavelin firmware to your hardware. This may take a few seconds to compile on a slower PC, and will be followed by a download plus programming and verification of Eeprom.

Please make sure that "F11" is used and not the more frequently used "F10" as the Firmware must be burned to Eeprom not run from RAM.

Leave the Propeller hardware powered-up and connected, and close the PropTool.

Checking PropJavelin Firmware Installation

Launch Javelin.exe. Select "Options" then "Debugger" and select the serial port used for program

download to the Propeller when using the Propeller Tool. Click "OK".

Click Project/Test Connection. This should show communications in the lower IDE panel and eventually result in a pop-up indicating "Javelin found on COMx".

Run Your First Java Program

Start Javelin.exe and load the previously downloaded TerminalTest2.java into the IDE. Click "Compile" or press "F9" to ensure the program compiles correctly; the status bar should show "Compile Successful".

Click Project/Download to RAM. The program will be compiled, linked and downloaded to the PropJavelin. A 'progress' pop-up will show during the process, and when complete the PropJavelin will be reset and the "Javelin Terminal" window will appear.

Re-size and adjust the splitter-bars as appropriate then in the bottom text box type "h", "e", "l", "p" and then press return.

The Terminal will show a series of "received character" messages and then "Unknown command : hheellpp".

PropJavelin is a work in progress and as can be seen above not everything works perfectly at this time - every character typed is sent twice - however it was the TerminalTest2.java program running on the PropJavelin itself which received and handled what was sent from the Terminal.

Debugging Your Java Program

Close the Javelin Terminal Window and close the PropJavelin IDE. Turn your PropJavelin hardware off and then back on.

Restart javelin.exe, re-load TerminalTest2.java, then click Project/Debug to RAM. The 'Progress' pop-up will appear and the Java program downloaded to the PropJavelin.

Once download is complete, the "Debugger" window will pop-up and the Java program can be single-stepped and otherwise debugged using the Debugger.

Notes

The PropJavelin project is a work in progress and incomplete at the time this document was written. A phenomenal amount of work was completed in just five weeks of the project commencing, a credit to Peter Verkaik and all those who have contributed to the PropJavelin project.

Further details of the ongoing project can be found in this thread. All contributions and help will be gratefully received. Knowledge of Java, the Javelin Stamp or PropJavelin is not a prerequisite or obstacle to participation.

When altering the Options/Editor configuration of the Javelin Stamp IDE or PropJavelin IDE it may be necessary to close and restart the IDE for all changes to be applied.

When using Project/Identify do not click on Refresh or Close while scanning is in progress. A PropJavelin when identified this way should show "Propeller, $78, 16384, No, No, Yes".

FEMTOBASIC

FemtoBASIC started out as a joke by Tomas Rokicki, a simple simulation of the Color Computer which could handle a couple simple commands. Mike Green took it to the next level incorporating many common BASIC commands as well as support for reading/writing to/from EEPROM and SD media.

Various spin-off's *(pun intended)* include the following:

FemtoBASIC The original FemtoBASIC, which continues to be improved upon by Mike Green
DongleBASIC An adaption created for use with Hitt Consulting's Propeller Dongle.
BoeBotBasic An adaption created for use with the BoeBot, supporting PING, IR,. and an HM55B compass.
PropterminalBASIC An adaption created for use with Propterminal, supporting Propterminal graphics commands.
FemtoBASICcolor An adaption incorporating AiGeneric drivers allowing 40x23, 16 color text.

FFT

As everybody knows, FFT stands for Fast Fourier Transform. It is commonly available in some sort or another for most microprocessors, and now the propeller is no exception!. The implementation shown below uses a sine/cosine table, an input buffer (for real and imaginary part) and outputs to the same input buffers.
Calculations are done using signed integer numbers. Multiplication is done using an unrolled Chip's algorithm with some improvements to make it signed-aware (using the **_sgn** variable). Bit reversal is performed using the convenient **rev** instruction, saving 75% of the time that a normal for/while algorithm will take. The butterfly loops are just standard without surprises, the variables are scaled for better use of the **adds** and **subs** instructions (signed math).

Spectrum of a run, using a base frequency by 5, like the test routine for int_fft.c describes

Input

The input should be an array of signed 16 bit values in the range -32768 to 32768. Scaling is applied automatically, so it would be a good idea to use the whole range. If your data is just 8 bit from an ADC, well scaling can be done for example in the decimation (bit-reversal) routine without incurring in too much overhead,
Only real data is used in this bit-reversal. But imaginary data could be also shuffled adding few instructions.

Output

The output is a 2 dimensional array (like the input), with the real part occupying the first half and the imaginary the second one.

Samples

This implementation works for a 1024 sample FFT. It takes around 4720000 cycles, 20480 multiplications, 5120 passes of the inner butterfly. Expected. That is enough to get 16 fps (@80 MHz). That is without the absolute value calculation (1024 multiplications, and 512 square roots, 532000 cycles) and plot drawing routines.

Code

The code shown below can be optimized a bit more, but it works !

```
.section cog cog0 ' needed to generate

' Converted to Propeller Assembler by Pacito.Sys, based on int_fft.c b
y Tom Roberts
' with portability by Malcolm Slaney.
' Distributed under the terms of the GNU GPL v2.0.
'
' Integer FFT
' 16 bit signed values are used


NN=1024
BITS_NN=10
BITS_NNM1=9
BITS_DIFF=3

init                    mov     fft_fr,cnt_rsample_ptr  ' real part bu
ffer, 2048 bytes
                        mov     fft_fi,cnt_isample_ptr  ' imag part bu
ffer, 2048 bytes

                        mov     fft_n,#1
                        shl     fft_n,#BITS_NN          '1024 point ff
t

                        call    #decimate
                        call    #lets_rock
                        call    #calc_abs
                        call    #plot

init_end                jmp     #init_end               ' end

' bit-reversal, uses the nice rev instruction
decimate                mov     fft_ii,#1
                        mov     fft_ll,fft_n
ldecimate               mov     fft_jj,fft_ii
                        rev     fft_jj,#32-BITS_NN     ' BITS_NN will
be reversed
                        cmp     fft_ii,fft_jj   wc
              if_nc     jmp     #ldecimate_5
                        mov     fft_fr_ii,fft_ii
                        mov     fft_fr_jj,fft_jj
                        shl     fft_fr_ii,#1
                        add     fft_fr_ii,fft_fr
```

```
                        rdword  fft_tr,fft_fr_ii
                        shl     fft_fr_jj,#1
                        add     fft_fr_jj,fft_fr
                        rdword  fft_result,fft_fr_jj
                        wrword  fft_tr,fft_fr_jj
                        wrword  fft_result,fft_fr_ii
ldecimate_5             add     fft_ii,#1
                        cmp     fft_ii,fft_ll   wc, wz
              if_c_or_z jmp     #ldecimate
decimate_ret            ret

' Calcs the 1024 point-FFT using 16 bit signed integers, some calculat
ions
' are don with 32 bits

lets_rock               mov     fft_ll,#1
                        mov     fft_k,#BITS_NNM1

lets_rock_while         cmp     fft_ll,fft_n      wc
              if_nc     jmp     #lets_rock_while_e

                        mov     fft_is,fft_ll
                        shl     fft_is,#1
                        mov     fft_m,#0
lets_rock_for_1         cmp     fft_m,fft_ll      wc
              if_nc     jmp     #lets_rock_for_1_e

                        mov     fft_jj,fft_m
                        shl     fft_jj,fft_k

                        call    #get_sincos
                        mov     fft_ii,fft_m
lets_rock_for_2         cmp     fft_ii,fft_n      wc
              if_nc     jmp     #lets_rock_for_2_e

                        mov     fft_jj,fft_ii
                        add     fft_jj,fft_ll

                        mov     fft_fi_jj,fft_jj
                        shl     fft_fi_jj,#1      ' word access
                        mov     fft_fr_jj,fft_fi_jj
                        add     fft_fr_jj,fft_fr
                        add     fft_fi_jj,fft_fi

                        rdword  fft_result,fft_fr_jj
                        call    #lets_mul_wr
```

```
                        mov     fft_tr,fft_result
                        rdword  fft_result,fft_fi_jj
                        call    #lets_mul_wi
                        subs    fft_tr,fft_result     ' 32 bit signed
value

                        rdword  fft_result,fft_fi_jj
                        call    #lets_mul_wr
                        mov     fft_ti,fft_result
                        rdword  fft_result,fft_fr_jj
                        call    #lets_mul_wi
                        adds    fft_ti,fft_result     ' 32 bit signed
value

                        mov     fft_fi_ii,fft_ii
                        shl     fft_fi_ii,#1          ' word access
                        mov     fft_fr_ii,fft_fi_ii
                        add     fft_fr_ii,fft_fr
                        add     fft_fi_ii,fft_fi

                        rdword  fft_qr,fft_fr_ii      ' qr = fr[i]
                        shl     fft_qr,#16
                        sar     fft_qr,#1             ' scales to 32 b
it signed value
                        mov     fft_result,fft_tr
                        rdword  fft_qi,fft_fi_ii      ' qi = fi[i]
                        shl     fft_qi,#16
                        sar     fft_qi,#1             ' scales to 32 b
it signed value
                        adds    fft_result,fft_qr     ' res = tr + qr
                        subs    fft_qr,fft_tr         ' qr = qr - tr
                        shr     fft_result,#16        ' scales down
                        wrword  fft_result,fft_fr_ii  ' fr[i] = res =
tr + qr

                        mov     fft_result,fft_ti
                        adds    fft_result,fft_qi     ' res = ti + qi
                        shr     fft_qr,#16            ' scales down
                        wrword  fft_qr,fft_fr_jj      ' fr[j] = qr = q
r - tr
                        subs    fft_qi,fft_ti         ' qi = qi - ti
                        shr     fft_result,#16        ' scales down
                        wrword  fft_result,fft_fi_ii  ' fi[i] = ti + q
i
                        shr     fft_qi,#16            ' scales down
                        add     fft_ii,fft_is
```

```
                        wrword  fft_qi,fft_fi_jj      ' fi[j] = qi = q
i - ti
                        jmp     #lets_rock_for_2
lets_rock_for_2_e
                        add     fft_m,#1
                        jmp     #lets_rock_for_1
lets_rock_for_1_e       sub     fft_k,#1
                        mov     fft_ll,fft_is
                        jmp     #lets_rock_while
lets_rock_while_e
lets_rock_ret           ret


lets_mul_wi             mov     fft_sgn,fft_result
                        and     fft_sgn,cnt_sgn     wz
                        shl     fft_result,#16
                        negnz   fft_result,fft_result
                        shr     fft_result,#15

                        shr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
              if_c      add     fft_result,fft_wi   wc
                        rcr     fft_result,#1       wc
```

```
              if_c      add     fft_result,fft_wi    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wi    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wi    wc
                        rcr     fft_result,#1        wc
                        xor     fft_sgn,fft_sgnwi    wz
                        negnz   fft_result,fft_result
lets_mul_wi_ret         ret

lets_mul_wr             mov     fft_sgn,fft_result
                        and     fft_sgn,cnt_sgn      wz
                        shl     fft_result,#16
                        negnz   fft_result,fft_result
                        shr     fft_result,#15
                        shr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_wr    wc
                        rcr     fft_result,#1        wc
```

```
              if_c      add     fft_result,fft_wr   wc
                        rcr     fft_result,#1       wc
                        xor     fft_sgn,fft_sgnwr   wz
                        negnz   fft_result,fft_result
lets_mul_wr_ret         ret

' Uses the ROM table to get the sine and cosine of jj

get_sincos              mov     fft_wr,fft_jj
                        shl     fft_wr,#BITS_DIFF
                        mov     fft_wi,fft_wr
                        add     fft_wr,cnt_sin_90
                        test    fft_wi,cnt_sin_90     wc
                        test    fft_wi,cnt_sin_180    wz
                        negc    fft_wi,fft_wi
                        or      fft_wi,cnt_sin_table
                        shl     fft_wi,#1
                        rdword  fft_wi,fft_wi
              if_z      mov     fft_sgnwi,cnt_sgn        ' they are in
verted
              if_nz     mov     fft_sgnwi,#0
                        test    fft_wr,cnt_sin_90     wc
                        test    fft_wr,cnt_sin_180    wz
                        negc    fft_wr,fft_wr
                        or      fft_wr,cnt_sin_table
                        shl     fft_wr,#1
                        rdword  fft_wr,fft_wr
              if_nz     mov     fft_sgnwr,cnt_sgn        ' they are no
t inverted
              if_z      mov     fft_sgnwr,#0

                        shl     fft_wr,#14
                        shl     fft_wi,#14
get_sincos_ret          ret

lets_mul_qr             mov     fft_result,fft_qr
                        shl     fft_result,#16
                        abs     fft_result,fft_result
                        mov     fft_qr,fft_result
                        shr     fft_result,#16

                        shr     fft_result,#1        wc
              if_c      add     fft_result,fft_qr    wc
                        rcr     fft_result,#1        wc
              if_c      add     fft_result,fft_qr    wc
                        rcr     fft_result,#1        wc
```

```
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1         wc
              if_c      add     fft_result,fft_qr     wc
                        rcr     fft_result,#1
lets_mul_qr_ret         ret

lets_sqrt_qi            mov     fft_result,#0
                        mov     fft_m,#0
                        mov     fft_jj,#16
lets_sqrt_qi_l          shl     fft_qi,#1   wc
                        rcl     fft_m,#1
                        shl     fft_qi,#1   wc
                        rcl     fft_m,#1
                        shl     fft_result,#2
                        or      fft_result,#1
                        cmpsub  fft_m,fft_result   wc, wr
                        shr     fft_result,#2
                        rcl     fft_result,#1
                        djnz    fft_jj,#lets_sqrt_qi_l
lets_sqrt_qi_ret        ret

calc_abs                mov      fft_ii,#511
```

```
                        mov     fft_fr_ii,fft_fr
                        mov     fft_fi_ii,fft_fi
calc_abs_5              rdword  fft_qr,fft_fr_ii
                        call    #lets_mul_qr
                        mov     fft_qi,fft_result
                        rdword  fft_qr,fft_fi_ii
                        add     fft_fi_ii,#2          ' next word
                        call    #lets_mul_qr
                        add     fft_qi,fft_result
                        call    #lets_sqrt_qi
                        wrword  fft_result,fft_fr_ii
                        add     fft_fr_ii,#2          ' next word
                        djnz    fft_ii,#calc_abs_5
calc_abs_ret            ret

' This routine will draw the spectrum in a 1bpp 320x240 bitmap

plot                    mov     fft_ii,#40
                        mov     fft_jj,#0

                        mov     fft_fr_ii,fft_fr
plot_8p                 mov     fft_k,#$80
                        rdword  fft_qr,fft_fr_ii
                        add     fft_fr_ii,#2
                        call    #putpix
                        shr     fft_k,#1
                        rdword  fft_qr,fft_fr_ii
                        add     fft_fr_ii,#2
                        call    #putpix
                        shr     fft_k,#1
                        rdword  fft_qr,fft_fr_ii
                        add     fft_fr_ii,#2
                        call    #putpix
                        shr     fft_k,#1
                        rdword  fft_qr,fft_fr_ii
                        add     fft_fr_ii,#2
                        call    #putpix
                        shr     fft_k,#1
                        rdword  fft_qr,fft_fr_ii
                        add     fft_fr_ii,#2
                        call    #putpix
                        shr     fft_k,#1
                        rdword  fft_qr,fft_fr_ii
                        add     fft_fr_ii,#2
                        call    #putpix
                        shr     fft_k,#1
```

```
                        rdword   fft_qr,fft_fr_ii
                        add      fft_fr_ii,#2
                        call     #putpix
                        shr      fft_k,#1
                        rdword   fft_qr,fft_fr_ii
                        add      fft_fr_ii,#2
                        call     #putpix
                        add      fft_jj,#1
                        djnz     fft_ii,#plot_8p

plot_ret                ret

putpix                  mov      fft_qi,#239
                        max      fft_qi,fft_qr
                        mov      fft_qr,#239
                        sub      fft_qr,fft_qi
                        shl      fft_qr,#3
                        mov      fft_qi,fft_qr
                        shl      fft_qr,#2
                        add      fft_qr,fft_qi
                        add      fft_qr,cnt_bitmap_ptr
                        add      fft_qr,fft_jj
                        rdbyte   fft_ll,fft_qr
                        or       fft_ll,fft_k
                        wrbyte   fft_ll,fft_qr
putpix_ret              ret

' constants
cnt_sgn                 long    $8000
cnt_sin_90              long    $0800
cnt_sin_180             long    $1000
cnt_sin_table           long    $7000
cnt_rsample_ptr         long    $800
cnt_isample_ptr         long    $1000
cnt_bitmap_ptr          long    $4000
cnt_add_ptr             long    512

' Variables

fft_ii                  long    0
fft_is                  long    0
fft_jj                  long    0
fft_k                   long    0
fft_ll                  long    0
fft_m                   long    0
fft_n                   long    0
```

```
fft_qr                  long    0
fft_qi                  long    0
fft_fr                  long    0
fft_fi                  long    0
fft_tr                  long    0
fft_ti                  long    0
fft_wr                  long    0
fft_wi                  long    0
fft_fi_ii               long    0
fft_fr_ii               long    0
fft_fi_jj               long    0
fft_fr_jj               long    0
fft_result              long    0
fft_sgn                 long    0
fft_sgnwr               long    0   ' sign of wr
fft_sgnwi               long    0   ' sign of wi
```

This can be adapted with minor modifications to 256 or 512 points or extended to 2048 or 4096 points, in those cases some constants need adjustment

- **NN** : Number of samples
- **BITS_NN** : Log2(Number of samples)
- **BITS_NNM1** : BITS_NN - 1
- **BITS_DIFF** : Difference between the amount of samples (divided by 2) and the samples in the sine table.

Some modifications to **get_abs** and **plot** may be then necessary.
Note: This was tested with pPropellerSim, as of today not yet available with all bugs fixed (**rev**, **max**, etc).

Enjoy !

Fixed Point Math

Fixed point Math makes reference to a way of number representation that allows fast calculations without all the burden of a full floating point implementation with some floating point advantages, i.e decimal places. This article will be focused in 16.16 Fixed point numbers (aka FP16.16).

This representation has important properties:

There is no exponent
The number is denormalized
The fraction does not represent decimal digits, is a binary fraction.

The number one (1.0) is represented as $1_0000
The number two (2.0) is represented as $2_0000
The number 1/2 (0.5) is represented as $0_8000

Arithmetic

Addition and subtraction of two FP16.16 yields directly a FP16.16

```
    add  n1,n2
n1  long $1_0000
n2  long $2_0000
```

yields n1 = $3_0000

Multiplication is a bit more complicated because the result gets scaled as a consequence of taking the number of bits of both arguments added.

```
    mull  n1,n2     ' inexistent 32x32 multiplication, returns high 32
 bits discards lower 32 bits
n1  long  $2_0000
n2  long  §3_0000

yields $6_0000
```

Division is a bit much more complicated, because scaling also occurs and all non-zero digits to the right of the decimal point will be lost:

```
    divl  n1,n2     ' hypothetic 32 by 32 bit division
n1  long  $5_0000   ' 5.0
n2  long  $0_8000   ' 0.5

will yield $0_000A, an unscaled result (should be $A_0000, 10.0)

    divl  n1,n2     ' hypothetic 32 by 32 bit division
n1  long  $5_0000   ' 5.0
n2  long  $0_4000   ' 0.25

will yield $0_0014, an unscaled result (should be $14_0000, 20.0)
```

((More to come))

Full Duplex Serial

(and other asynchronous serial variations)

Full Duplex Serial	Created by: Chip Gracey	The original "Full Duplex Serial"
Extended Full Duplex Serial	Created by: Martin Hebel	Extensions to allow reception of character strings that end in a carriage return for decimal, hexadecimal and alpha-numeric uses. Allows use of timeout values. This version also allows use of defined delimiter characters, and returning whole and fractional portions of a numeric string.
Simple Serial	Created by: Chip Gracey	Bit-bang serial driver for low baud rate (~19.2K) devices.
SerialMirror	Created by: Mirror	1. This driver is designed to be method-compatible with the FullDuplex serial object, allowing it to be used when high speed comms or devoting an independent cog for serial I/O is not necessary. 2. Bi-directional communication on the same pin is also supported. A FullDuplexSerial enhancement that allows a single serial connection to be used from within multiple spin files. Primarily aimed as a debugging aid, this file also illustrates how to use the DAT section to make single-instance objects which may be called from multiple spin files.

Designed for serial communications.

Examples of this code in use: PropCOMM,

.start(rxpin, txpin, mode, baudrate)
Start serial driver - starts a cog
mode bit 0 = invert rx
mode bit 1 = invert tx
mode bit 2 = open-drain/source tx
mode bit 3 = ignore tx echo on rx

.rx(rxbyte)
Receive a byte of data

.tx(txbyte)
Send a byte of data

.str(stringptr)
Send string. Example: *str(string("Hello World"))*

.dec(value)
Print a decimal number

.hex(value, digits)
Print a hexadecimal number

.bin(value, digits)
Print a binary number

Game Programming for the Propeller Powered Hydra

by André LaMothe

ISBN 1-928982-40-9

An 812 page book which comes included in the box with the Hydra Console. It is also for sale separately. It has an accompanying CD with a few games, lots of starters for games, other demos and some Graphics Drivers. The software on the CD is for use by the owner. It is not open source. Drivers from the CD must not be distributed with games.

You can download some sample chapters, useful for understanding the Parallax reference graphics drivers.

Errata:

p. 648 Some clones of Defender were tile-based as stated. However the original William's Defender used a bitmapped frame buffer.

Playable Games

These are mostly games for the Hydra. Many are adaptable for other Propeller boards.

Game	Picture	Description	By	Location
	(Use external image where possible to save space)			
Aliens Invaders		A vertically scrolling shoot 'em up	Remi Veilleux	On Hydra Book CD
Ball Buster		A Breakout clone	JT Cook	On Hydra Book CD
Defender		A Defender Clone	Game by Steve Waddicor Sound by Eric Moyer	Parallax forums
Dodgy Kong		A Donkey Kong Clone	Game by Steve Waddicor Sound by Eric Moyer	Dodgy Kong zip
Dr Hydra		A Dr Mario Clone	Remi Veilleux	On Hydra Book CD
HYDRA Lock n Chase		A remake of the classic "Lock 'n' Chase"	Remi Veilleux	On Hydra Book CD
Hydra Repeat		Simon clone	Spork Frog	Located in Parallax forums
Manic Miner		A Manic Miner clone	Jim Bagley	Manic Miner zip
Planetary Defense		Defend your planet from missiles and aliens by shooting them down	Mpark	Located in Parallax forums
SpaceWar!		A Spacewar! clone	Eric Moyer	First post in the SpaceWar! thread
Spintris		A 1 or 2 player Tetris clone. NTSC	JT Cook	Located in Parallax forums

	and PAL		
X-Racer	A car racing game in the style of Pole Position	JT Cook	On Hydra Book CD

The following was originally published July 11, 2011 at
http://blog.2e-pro.com/2011/07/why-is-logo-important.html
When you are an Internet marketer, you are your own brand. Some online marketers use their first and last name as their business name, while others actually choose a company name to go by. Either way, you have an online brand and you should also have a logo. When it comes to business logos, no matter what type of business the company or person is in, there are some logos which fit the bill and others that are simply lacking.

When it comes to Internet marketing, your logo is very important and if you do not have a good logo, then it can easily affect your marketing success. Here are 4 reasons why:

Brand
A website's logo often becomes attached to its brand. It is what people think of when people think about a website, an online business and the person running the business. It can also make people remember a person's name easier if their name is attached to the logo in some manner. If the logo is not catchy or is poorly done, then the marketer could risk having people not remember the website, what the website is about, the marketer and the company as a whole.

Recognition

When your logo is done well and people remember it than they remember you and your company. The more times and the more places that they see your logo, the more they will associate with you and your business.

Trust

A person, company and website that is using the same logo (if well done) across all of the networks (website, social media accounts, business cards, letterhead, email signature etc.) gains trust with the consumer. Consumers like it when everything is the same across the board. If a marketer or online company is using different logos on different accounts or not using a logo that looks nice and is catchy than that could decrease consumer trust or prevent the consumer from trusting the brand in the first place.
Attention

Last, a logo that is nicely done brings attention. Businesses online or off are often awarded for their logo design and it can bring attention from other markets. The more attention a company has, the better.

Take a good hard look at your logo or think about developing one if you do not already have one. Remember, if you have had the same logo for several years and you have a strong following, then you may not want to change it even if it is lacking some elements. Think how changing the logo hurt the clothing company GAP last year and the consumer backlash against it. You do not want to go down that road but if your logo has not been around that long and you are just building up your Internet marketing company, it is better to get your logo right now.

By ITM Marketing Blog
'The Internet Time Machine'
www.theinternettimemachine.com

Graphics Drivers

Tiles and Sprites drivers

Driver name	Location	Resolution (WxH)	Tiles	Sprites	Color Palette	Cogs	Source files
COP	On Hydra Book CD						cop_drv_010.spin
DK	As part of latest Dodgy Kong zip	256x224	32x28 map 8x8 pixels/tile	34 total Number per line depends on cogs used 16x16 pixels	4 colors per individual tile 4 colors per individual sprite	3+	sw_dk_gfx_renderer_015.spin sw_dk_tv_drv_022.spin
HEL	On Hydra Book CD	160x192?	10x12 map 16x16 pixels/tile Horizontally scrollable	8 total 5 per line 16x16 pixels Sprites share palettes of tiles they overlap with	4 colors per individual tile	1	HEL_GFX_ENGINE_050.spin
JLC Spectrum	Parallax Hydra forum	256x192	32x24 map 8x8 tiles	No sprites.	2 colors per individual tile chosen from a palette of 16 plus flashing colors.	1?	JLC_Spectrum_TV_010.spin
REM	On Hydra Book CD						rem_gfx_engine_017.spin rem_tv_017.spin
VGA Learning Driver	Propeller forum	640x480	N/A	N/A	64 color (4 color / 16pixel mode)	1	VGA Learn.spin

NTSC Tutorial	Hydra forum	188x244	N/A	N/A	N/A	1	BAM_50_Line_Driver_01.spin
8x8 Tile Driver Tutorial	Hydra forum	376x240	30x64 tile map, 8x8 tiles, smooth scroll in X and Y, rows can be added but need to be at least 30.	Any height, width can be 4, 8, 16, 32, 64, etc. pixels. Add sprites until the driver chokes, then add more cogs... ;-)	One byte per pixel, use the Hydra palette.	5+	See post in forum.

Character mode driver

TV_Text is in the standard library. PAL/NTSC support, 40 columns, 12 lines. or 16 lines for NTSC and PAL respectively.

8x8 NTSC driver (up to 70 chars / line, 25 display lines 2 colors / screen or up to 40 chars / line, 25 display lines 2 colors per char displayed)
This driver was further developed in this thread, into 2 versions plus some derived works to make it compatable with TV_Text. Look for the downloads from Hippy for the merged versions that have added easy to use Pin and mode selection.

Note that there is a improved graphics driver SDM_graphics_XOR, that is an extension to the standard graphics.spin to support XOR mode that can replace double buffering, and uses the internal fonts. This reduced memory usage might mean you do not need a text mode driver.

raster-based, non-tiled graphics system(s)

- "Amazing Sand Physics demo uses 6 cogs to animate 10,000 grains of sand in realtime"
- Variable resolution simple bitmap high-color NTSC driver (horizontal 20, 40, 80, 160, 256, 320 pixels Vertical 48, 64, 96, 192 x 1 byte / pixel)

Here is various information about the Propeller architecture and various boards that have a Propeller on them.

Propeller P8X32 die

Architecture Information

The Propeller P8X32 chip is a multicore microcontroller. It has 8 cores, called Cogs, 32k of shared RAM, 32k of ROM, and 32 I/O pins.
The Cogs each have their own local memory, and share access to the 32k Hub memory. They also share access to the 32 I/O pins. They, also, each have their own pair of user configurable counters with PLLs and a Video Generator that aids in driving either NTSC/PAL or VGA video outputs.
Hub RAM - 32K Ram shared by the Cogs
Cog RAM - 512 Longs local memory per Cog
Oscillator - Options and Over-Clocking
Interrupts - Handling 'external signal events'
Video Generator - Output Video signals
Propeller Block Diagram PDF
Propeller Data Sheet v1.2 PDF

Development Boards

- SerPlug an inexpensive alternative to the PropPlug - program your propeller from a serial port or USB to serial cable.
- Chameleon PIC/AVR a multi-processor "Arduino Like" system consisting of either a Microchip PIC24 16-Bit or an Atmel AVR 328P 8-Bit microcontroller and a Parallax propeller.
- Propeller Demo Board included in the Propeller Starter Kit
- Professional Development Board USB & Serial versions
- Proto Board & Proto Board USB
- Education Kit 40 pin DIP & PropStick USB versions

- PropRPM Rapid Prototyping Module
- P8X32 Education Board
- PropStick USB
- Spin Stamp a Propeller chip in a stamp-like package
- HYDRA Game Console (HGC) included in the HYDRA Game Development Kit
- SpinStudio development board and various plug-in modules that can be used with other currently available Propeller development products.
- HYBRID Development Kit available as an unassembled kit or fully assembled board.
- Hive board a three-propeller design with ethernet, SRAM, SD card slot of a mostly german speaking community. Board is available as kit.
- The Briel PockeTerm available as a fully assembled board, a complete kit, or just blank PCB.
- Propeller Platform Arduino-style modular system.
- Propeller Platform USB Same as the previous, but has some upgrades such as MicroSD slot and USB.
- Propeller System Module All-in-one with LCD display, accelerometer, SD Card Slot, and USB.
- Morpheus: Dual Propeller SBC with 512KB RAM, 1MB Flash, RTC, 256 color VGA and expansion bus, Mem+ 2MB memory/IO expansion board with RS232 port & programming, SD card support and 16 bits digital I/O
- Propteus: Propeller Prototyping Board with support for DIP Propeller, EEPROM, PropellerPlug compatible header, Reset switch, optional 4 user LED's,large prototype area, stacks on Morpheus or other Parallax boards, 12 servo headers, bussed large prototyping area.
- Proteus: A bussed prototyping board meant to mount on Morpheus, Propteus, Propeller Proto Board (serial and usb version) or any 4"x3" Parallax style board.
- SmorgasBoard 2012: A collection of 23 open source propeller designs in a 11x8", tab routed PCB panel

I/O Bus Systems for some of the above listed development boards.
Development Board Differences- Xtals and pin usage differences between some of the above listed boards.

Other development boards using the propeller and another processor

- pPropQL board. A hardware emulator of the Sinclair QL (using a MC68008)
- pPropQL020 board. An extended version using a MC68EC020 microprocessor
- pProp040 board. An improved version of the 020 board but with a 040 :).

Propeller

This site is dedicated to documenting interesting stuff related to the Parallax Propeller microcontroller.

Software

A bunch of information about programming the Propeller.

Development Tools

Tools to help you in developing software for the Propeller chip.

Hardware

Information about the Propeller Chip and several Development Boards.

Interfacing

Connecting the Propeller to various devices.

Released projects

Several software projects available for the Propeller.

Propeller Magazine

Online magazine with lots of links to interesting stuff about the Propeller. (no longer updated)

- Propeller Lingo- with some keywords about the propeller, and what it all means.
- Where in the World are other users and Propeller related companies.
- Links to other sites related to the Propeller.
- #Propeller on IRC - Chat with other Propeller users in real time

---

Rumoured information about the next generation Propeller II (Updated as new information is revealed)

This will only grow into something worthwhile if people contribute to it. Please feel free to add new material, or edit and improve what's here. If you don't feel confident enough to edit a page, or if you have a question or suggestions for improving a page, add a comment on the relevant discussion page.

Homespun is a command-line Spin compiler that reads .spin files and outputs .eeprom files. It is written in C# and runs on Windows with the .Net framework and on Linux with Mono.

Homespun is a personal project and not a professional product. Use Homespun at your own risk.

Download the executable here:Propeller forum thread.

In my testing, Homespun generates output that is identical to Proptool's (except the occasional least-significant bit in floating-point constants), which was my original goal. Now I'm using Homespun as a testbed to try out extensions to the compiler and the language. All information here is subject to change as Homespun continues to evolve.

Installing Homespun

Installation consists simply of copying homespun*XXX*.exe (where *XXX* is the version number) to your Windows (or Linux) machine. As long as you have the .Net framework (or Mono) installed, Homespun should run.

Using Homespun in Linux under wine

There are two ways of running homespun. One is directly from a terminal using wine homespunXXX [options] . The other one is using a wine terminal, for this the command has to be mono homespunXXX.exe [options] . This seems to be a limitation of mono (at least in the 1.9.1 and 2.2 versions).

Using Homespun in Mac OS X under Darwine

There is no wine executable that you can easily call (and I'm aware of) from the Mac OS X terminal. So the way to get homespun to run is to use a Darwine terminal settings file. This file is a settings file for the Mac OS X terminal that will launch the command line interpreter in darwine (located in /Applications/Darwine/Wine.bundle/Contents/bin/wine). From this command line the path to the mono executable can be added, but it must be done every time:

```
set path=C:\Program Files\Mono-2.2\bin;%PATH%
```

A more permanent way is to add it to the registry in HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Environment\PATH

```
PATH=C:\windows\system32;C:\windows;C:\Program Files\Mono-2.2\bin
```

Every time you launch the new terminal properties file this path will be set so you can launch homespunXXX.exe [options] . This seems to be a limitation of mono (at least in the 2.2 version I tested under darwine).

Using Homespun

At a command prompt, type "homespun*XXX filename*.spin". You can omit the ".spin"; Homespun will add it. The program will compile the Spin source code and produce a file called *filename*.eeprom (or *filename*.binary).

Command-line options

You may specify command-line options and the filename in any order. Options begin with "-".

- **-b** - generates a .binary file instead of the default .eeprom file.
- -**c** - generates a .dat file containing the top object's DAT data.
- **-d** - produces a listing file, including disassembled Spin bytecode.
- -**D** - define a symbol (as in #define); e.g. -D DEBUG
- **-i***n* - controls informational messages: **-i0** suppresses all messages, **-i2** and **-i3** show some messages, **-i3** shows all messages.
- **-L** - specify library path. See below for more information.
- **-w** - enables warnings for common coding pitfalls -- code that is syntactically correct but perhaps does something other than what you intended. See below for more information.
- **-sob** - generates a .sob file for every .spin file compiled.

Object sharing

Details to come. *Or maybe not. Is anybody reading this? Email me if you're interested.*

Warnings

If the **-w** option is specified, Homespun will generate a warning when it encounters the following potentially problematic code:

- JMP/DJNZ/etc without #
- RES without a count
- ORG without an address
- data after RES
- COGNEW containing a call to another object
- data truncation (e.g. DAT BYTE 1000)

Library search paths

Filenames in OBJ declarations and FILE directives can contain ":" and "\" (and "/"), so you can hardcode full paths. However, a more flexible approach is to use the **-L** option and the SPINLIB environment variable. Invoke Homespun using **-L** to specify directories (one **-L** per directory). For example,

```
Homespun028 -L \second\ -L ..\third\ first\blah.spin
```

If your input filename contains a directory, as in this example, that directory is searched first. If you just supply the input filename, the current directory is searched first.

In this example, Homespun will search for blah.spin first in first\, then in \second\, then ..\third\. Homespun will search those same directories for any objects referenced in blah.spin.
(Note: Homespun will supply the trailing "\" if necessary. It will also switch to "/" on non-Windows systems, but to be on the safe side, add the final separator yourself.)

You can also set the SPINLIB environment variable to specify additional directories:

```
set spinlib=c:\fourth\;\Program Files\Parallax Inc\Propeller Tool v1.0
5.5
```

The **/L** option takes precedence over SPINLIB, so if we invoke Homespun with the same command line as before, the search order will be

- first
- \second\
- ..\third\
- c:\fourth\
- \Program Files\Parallax Inc\Propeller Tool v1.05.5\

Multidimensional arrays

VAR variables can now be multidimensional arrays (any number of dimensions).

```
VAR byte myArray[2,40,13] ' a 2x40x13 array
```

```
... in a PUB or PRI:
  myArray[i,j,k] := blah
```

No bounds-checking is performed.

#include directive

You can include a file inside another file using #include. Example:

```
#include "defs.h"
```

Homespun will search the library paths for the file.

Conditional compilation

Homespun supports the following conditional compilation directives:

- #ifdef *symbol*
- #elseifdef *symbol*
- #else
- #endif
- #ifndef

#ifdef and #elseifdef check to see if *symbol* has been defined using #define (see below).
Example:

```
#define HYDRA
CON
#ifdef HYDRA
  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000
#elseifdef HYBRID
  _clkmode = xtal1 + pll16x
  _xinfreq = 6_000_000
#else
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000
#endif
```

```
#ifndef MYDEF
#define MYDEF
#endif
```

Text substitution macros

You can do simple text substitution using #define. The syntax of the #define directive is:

```
#define <symbol> <replacement-text>
```

is whatever text follows on that line.
Any identifier tokens in the source file that match will be replaced with . Matching is case-insensitive.
For example:

```
#ifdef PropII
#define ADDR LONG
#else
#define ADDR WORD
#endif

var addr baseaddress ' replaces addr with long or word, but doesn't to
uch the "addr" in baseaddress.
```

Note that the scope of a #defined symbol is from the #define to the end of the file. Like BST, Homespun passes #defined symbols down to sub-objects (provided the #define precedes the OBJ section).

Also note that all #-directives must start in column 1.

_SPACE constant

Homespun can reserve space in low hub memory (starting at $0010). Declare "CON _SPACE = xxx" where xxx is the number of bytes you want to reserve. Homespun totals up the _SPACE constants in all the objects in your program.

@@@ operator

@@@*datSymbol* evaluates at compile time to the absolute hub address of *datSymbol*. *datSymbol* must be

defined in a DAT section.

Known bugs

- -d listing is incorrect if _SPACE is in effect.
- RESULT keyword is not handled correctly.

Planned work

- Make error messages less cryptic
- Clean up and release source code (a long way off)

Homespun is a work in progress. Feel free to contact me to report bugs or to suggest improvements. Many thanks to those who already have done so.

Author

Michael Park (mp__-at-hotmail.com)
Edit Feb 26, 2011 by Jazzed

Hub Memory Map

Hub memory contains 65536 bytes (or 32768 words, or 16384 longs). Hub Ram uses half of this space ($0000-$7FFF) and ROM uses the other half ($8000-$FFFF).

$0000	LONG ClkFreq	This value is created by the spin compiler, and is equivalent to the system variable _CLKFREQ that may be set in the Spin Top Object. If a user program changes the frequency using the CLKSET command, it should also update this value. Default is 12,000,000 which is the approximate frequency of RCFAST mode.
$0004	BYTE Clk	Equivalent to the system variable _CLKMODE that may be set in the Spin Top Object. If a user program changes the frequency using the CLKSET command, it should also update this value. Default is 0 which corresponds to RCFAST mode.
$0005	BYTE Checksum	
$0006	WORD TopObjectBaseAddress	Start address of the top object in a spin program. PropTool currently always sets this to $0010
$0008	WORD VariablesBaseAddress	Start address of Spin variables.
$000A	WORD StackBaseAddress	Start address of stack. Stack grows upwards from here.
$000C	WORD Main	Address of first PUB method in top object. Program execution starts here.
$000E	WORD InitialStackPointer	Initial value of stack pointer. Live stack pointer is kept in the interpreter Cog RAM, so this value doesn't get updated.

$0010-	Top object	Usual start of top object.
$8000-$BFFF	Character Set	256 characters of 16x32 (width x height) pixels in two colours. In fact that isn't the whole truth. The "characters" numbered 0, 1, 8, 9, 10, 11, 12, 13 are really eight patterns of 16x16 pixels in four colours. A fancy video driver can give a coloured background to a rectangle of 32x16 characters, and use the 16x16 patterns to add beveled edges and corners to the rectangle. The patterns also provide underlines, overlines, and triangular marks inside the corners, made visible or invisible by changing one of the four colours.
$C000-$CFFF	Log Table	2048 word values
$D000-$DFFF	Anti-log Table	2048 word values
$E000-$F001	Sine Table	2049 word values
$F002-$F003	Unused (padding)	The sine table contains word values while the Spin interpreter and boot loader are stored as long words. They must be aligned as described in Hub Ram. The change in alignment creates this unused space.
$F004-$F7C3	Spin Interpreter	To start the Spin interpreter, a cog is initialised with $F004 as the entry point and PAR pointing to $0004. The hardware copies 496 long words from this address range into the new cog's Cog Ram. The hardware loads the rest of the Cog Ram with zeros.
$F7C4-$F7FF	Unused	These long words are part of the space allocated for the Spin interpreter but are not copied.

$F800-$FFBF	Boot Loader	On reset, the hardware copies 496 long words from this address range into cog 0's Cog Ram. $FBB4-$FFBF (cog $0ED-$1EF) are copied but the boot loader doesn't use them in any way. $FF00-$FF5F (cog $1C0-$1D7) contain a copyright notice.
$FFC0-$FFFF	Unused	These long words are part of the space allocated for the boot loader but are not copied.

See Also

Hub Ram
Cog Ram

Hub RAM

Hub RAM is the primary memory of the Propeller chip. It is entirely separate from Cog RAM.

A program running in a Cog (whether a user program or the Propeller Spin Interpreter) has access to the Hub RAM through the use of RDLONG, RDWORD, RDBYTE, WRLONG, WRWORD and WRBYTE instructions. The design of the Propeller is such that all of these instructions are 'atomic', which means that they will always complete in full, never partially, and if two Cogs do write to the same Hub RAM location, the value placed will be that of the last written. The value placed will never be a mix of what both Cogs attempted to write. Use of Locks can be used to prevent non-atomic access to larger areas of Hub RAM.

The Hub RAM may be considered to be 32K x 8 bits, 16K x 16 bits, 8K x 32 bits or in any combination as circumstances dictate. To use RDLONG and WRLONG within a Cog, the address must be on a long boundary (the two lsb's must be zero). To use RDWORD and WRWORD the address must be on a word boundary (the lsb must be zero).

Hub RAM is loaded from external 32K x 8 bit I2C Eeprom after Reset or downloaded into from the Propeller Tool (or other third-party Propeller Bootloader). Once the Hub RAM has been loaded, a Spin Interpreter is loaded into Cog 0 which begins execution of the Spin program which has been loaded.

The external Eeprom and any downloaded program must always contain a Spin program to be executed, even if this is just a small Spin program to place the Propeller in an operating mode other than as a Spin interpreter.

This can be better understood when you recall that the Hub RAM does not contain primary machine code executed by the real processor as in conventional microcontrollers. The Hub RAM contains just - data. The meaning of them depends on what the real program(s) running in the COG(s) think appropriate. As COG 0 starts with the SPIN interpreter, it is the master to decide.

See alse:

Hub Memory Map

Forums with discussion about the Hybrid can be found here.
Hybrid Technical Manual PDF

Hydra Game Console

The Hydra Game Development Kit includes:

- The HYDRA Game Console with 128K EEPROM and a plethora of I/O interfaces.
- 9V DC Power Wall Adapter.
- PS/2 Mouse.
- PS/2 Mini Keyboard.
- Nintendo Compatible Gamepad.
- A/V Cable.
- USB Programming Cable.
- 128K Re-Programmable Game Card to Store Games and Applications.
- Blank "Experimenter" Card to Design Your own Add-On Hardware.
- "Game Programming for the Propeller Powered HYDRA" hard copy book by Andre' LaMothe.
- CD-ROM with all source, demos, and development tools.

Available from:

- Nurve Networks
- Parallax

Using the Propeller as an I2C Slave Device

AiChip Industries has released code which allows a Propeller to be used as a simple I2C Slave device.

Original Parallax Propeller Forum thread : here

Vetson 005 (2008-04-13) source code can be downloaded from the Parallax Forum : here - It is recommended to check the forum thread for later versions.

A selection of source code is provided -

- I2cSlave_Demo : Test code to proves it all works
- I2cSlave_Slave : The actual I2C Slave handler (PASM)
- I2cSlave_Master : I2C Master routines (Spin)
- I2cSlave_16Kx8Ram : Test of 16Kx8 I2C Ram
- I2cSlave_32Kx8Ram : Theoretical 32Kx8 I2C Ram
- I2cLogger_Demo : Demonstrates I2C Bus capture / logging

The I2C Slave implementation allows a number of I2C Ram devices to be emulated, limited only by Propeller Chip memory capacity and Cog Numbers. The entire 32KB Hub memory can be used. A 1Kx8 I2C Ram can be emulated per Cog without requiring any Hub resources. The maximum capacity for a single Propeller Chip is 39KB (one 32KBx8 plus seven 1Kx8).

A Propeller Chip I2C Ram is accessed in exactly the same way as an I2C eeprom would be. The I2C Slave supports 7-bit and 10-bit device addressing and is written in Propeller Assembler (PASM) for maximum speed. The I2C Ram is cleared at start-up.

In addition to the I2C Slave code a modified version of it is provided to demonstrate how I2C Bus capture and logging could be performed.

Integer GPS navigation

This page will discuss a little bit of GPS navigation, and a possible approach using integers only and cartesian planar projection.

A little bit of background

While not propeller related a little bit of background will help you deciding wheter the approach can suit your needs.

GPS System and Earth models

GPS (global positioning system) is a clever triangulation system that uses satellites.
GPS returns latitude-longitude coordinates in a system called WGS84.
Earth shape is extremely complex, and every mapping system must approximate the real shape. The most used shape is an ellipsoid, where the poles are on the shorter axis. The parameters of the ellipsoid are varying, and it is a matter of trade off.
Maps use projections of the curved shape over a plane. Projections must accept a trade off, because it is not possible to preserve distance, angles and area at the same time.
To have an idea of the many solution adopted is possible to go to spatialreference
Moreover sea level is dependent on measures based on the gravity. So the real sea level can differ a lot from the ellipsoid zero.
Wikipedia geoid shows a blue/red image of the difference between geoid and ellipsoid.

Navigation formulas

In order to calculate navigation data it is possible to use the Aviation Formulary.
Another source of formulas is movable type scripts.
While it simplifies the earth model is still requires heavy math, so a lighter solution can be used with the Propeller.

Assumptions used in this document

While the earth shape is extremely complex it is possible to heavily simplify and still get very good navigation data.

- Earth is a sphere
- 1/60th of degree on latitude (N/S coordinate) equals to 1 nautical mile
- We can accept a small error in the calculation

- We will stay away from the poles (in case an ad hoc planar projection can be used for near pole navigation)
- The next way point will be not too far (up to few hundred miles could be ok)

The planar projection

With planar projection the points on the sphere are projected on a plane tangent to the earth
Moreover:

- 1 Prime difference (1/60th of degree) in latitude is one nautical mile (1nm)
- 1 Prime difference (1/60th of degree) in longitude is 1nm*cos(avg_latitude), where avg_latitude is the average latitude
- The latitude projects to a vertical cartesian axis
- The longitude projects to a horizontal cartesian axis

So we moved from a spherical system to a planar, cartesian system. Calculations are much easier and requires less math.
On the other hand the model introduces errors and approximations.
Let's try to quantify the error with respect to the results obtained from the aviation formulary. We'll compare the results of our calculations with the distance between two points calculated with the great circle (shortest path, variable heading) and rhumb line (constant heading, in general longer path).

Point1-lat [degrees]	Point1-long [degrees]	Point2-lat [degrees]	Point2-long [degrees]	Dist (great circle)	Dist (rhumb)	Heading (rhumb)	Dist (cartesian)	Heading (cartesian)	Gerat Circle Distance Error [%]	Rhumb Heading error[°]
45.5	10.5	45.51	10.51	0.7327	0.7327	35.02	0.7327	35.03	0.0029	0.0024
45.5	10.5	45.51	10.5	0.6000	0.6000		0.6000			
45.5	10.5	45.5	10.51	0.4206	0.4206		0.4206			
45.5	10.5	45.6	10.6	7.3249	7.3249	35.00	7.3271	35.03	0.0293	0.024
45.5	10.5	45.6	10.5	6.0000	6.0000		6.0000			
45.5	10.5	45.5	10.6	4.2055	4.2055		4.2055			
45.5	10.5	46.0	11.0	36.5815	36.5816	34.91	36.6353	35.03	0.1469	0.12
45.5	10.5	46.0	10.5	30.0000	30.0000		30.0000			
45.5	10.5	46.0	11.0	36.5815	36.5816	34.91	36.6353	35.03	0.1469	0.12
40.0	10.0	50.0	15.0	636.0251	636.1264	19.40	642.5062	20.96	1.0031	1.56

So for short distances the errors are negligible. Near the poles the errors are bigger, nonetheless the errors are still small for distances up to few tens of nautical miles between waypoints.

Formulas

If you think this is your case let's move closer to the propeller, and try to use integers only.
The propeller is a 32 bit processor, so a variable can be 32 bit, or 4 bytes, or a long. A typical signed long in the propeller ranges between -2^32/2 and (2^32)/2-1, or -2,147,483,648 to 2,147,483,647.
The range we expect to handle in our system is -180 to 180 degrees (for longitude). If we move to prime the range is -10,800 to +10,800. So the idea is to add fraction of prime resolution as long as the propeller can hold.
We can arrive up to -1,800,000,000 to 1,800,000,000, which means we can get as fine as hudreths of tousanths of prime, or if you prefer 1/100000th of a nautical mile, which is less than a inch or less than 2 centimeters.
For example, if we have coordinates from GPS:
4512.3456 N (45° 12.3456' latitude)
01012.3456 E (10° 12.3456' longitude)
We will convert the this way:
integer latitude = (45*60+12.3456)*100000 = 271,234,560
integer longitude = (10*60+12.3456)*100000 = 61,234,560
If the coordinates are S and W the sign will be negative.

Range limits:

The max longitude is 180° -> 1,080,000,000 less than the range limit of the long.
The problem is when calculating a difference in latitude. While the max difference is 180°, if we have two points at 179°E and 179°W, we get the following coordinates as integers:
long1 = 179°W = -1,074,000,000
long2 = 179°E = 1,074,000,000

delta_longitude = long2 - long1 = 1,074,000,000 + 1,074,000,000 = 2,148,000,000 > max long range!!

Note that if delta_longitude > 1,080,000,000 (180°) then
delta_longitude = sign(-delta_longitude)*(2,160,000,000 - |delta_longitude|)
where 2,160,000,000 is 360°.

In our case delta_longitude = -12,000,000 which is within the range.
A possible workaround is to divide by two the coordinates, calculate the difference, and then multiply times 2. In case of the GPS strings used for example we have plenty of resolution available and not used (a factor of 10), so in this case the operation would be lossless.

The propeller can be connected to lots of different things.
NOTE: Links on this page have to be updated. Google searching with the old url generally leads to the updated link.

- The humble LED
- TV screen -- see NTSC Palette Mode
- VGA monitor --
- Controller-less Monochrome LCD
- "the safest way to interface a 5 V signal to the Propeller": try a 1 kOhm resistor, which limits current (and voltage) to safe amounts. For those rare cases where that isn't fast enough or powerful enough, see interfacing 3V and 5V devices.
- USB
  - To connect to a host PC -- " "Prop Dongle" Protoboard 1.5" x 2.75". Available"
  - Using the Propeller as a USB Slave (full duplex serial)
  - Using the Propeller as a USB Host
- SPI to and from flash memory cards: ...
  - Out-of-order SD card access
  - Trying to understand the SPI object.
  - Using MicroSD card
  - the SPI Engine
  - "booting" from a SD card
  - Which SD-MMC Card ? compares various flash memory card interfaces, DRAM (!), and FRAM.
  - "SD bootloader -- a very simple EEPROM-resident program that would allow you to pick an image file from an SD card, transfer it to RAM, and boot it. That's it -- no frills, no drivers, just a bootloader."
  - "high-speed reads and writes to an MMC card."
- SPI to and from other devices ... such as the MCP3208 and ADS1271 ADCs... *how?*
  - "Fastest Serial Speed?" suggests that it may be possible to "send" up to about 8MBaud. The fastest actual implementation: "Viewport v1.1 transfers data between pc host and propeller at 2Mbps using a single cog... full duplex."
  - ADC0831 ADC
  - PropNIC - Interface to a ENC28J60 - connect to a local LAN or run a webserver from a Propeller!
  - MCP2515 - Interface to CAN Bus controller from a Propeller (schematic with object at exchange) - See PropCAN web for pictures (Demo Board CAN I/F, PropCAN) Follow the development of PropCAN at turning on propCAN an engineering notebook Blog.
- SPI combinations
  - Read data from an A/D converter using SPI at 8000 samples per second; write to SD card
- I2C
  - I2C 12bit ADC for the Prop
  - Using the Propeller as an I2C Slave Device
- Serial ("RS232")
  - Two-Resistor Serial Interface
- RFID

    - RFID - With simple hardware
- Misc
    - multiple props
    - PWM - Pulse Width Modulation
    - TCP/IP Stack for the Propeller
    - hobby RC servo motors
        - hook the 3 wires up to: +5V and ground (a common ground with the propeller), and a 1 kOhm resistor from the propeller output to the servo input (as discussed above)
        - how many servos can be controlled by one Prop?
        - obtaining servo position
    - DMX - a serial protocol for controlling professional stage lighting devices.
    - Should we give a brief summary of the "received wisdom" of the "peripherals and interfacing" section of the "Good thread index"
    - Interfacing the propeller to other microprocessors (MC680X0): software OMU, hardware pPropQL and pPropQL020

Interrupts

The Propeller does not use interrupts as most other microcontrollers do; all external signalling events must be detected by polling or by waiting for an incoming line to go high or low. With the Propeller's multi-Cog architecture this is not as much of a problem as it would be for a single core processor; with one or more Cogs dedicated to 'interrupt handling', idling until the appropriate signal conditions are met, other Cogs can continue processing unaffected.

Multiple interrupts where response latency is not critical can be handled by a single Cog polling each 'interrupt line' in a cyclic manner and responding as appropriate. Where low latency is required a single Cog can wait for the interrupt to occur and respond within just a few clock cycles. Such a Cog can only handle a single interrupt, but with multiple Cogs that again is not so much of a problem.

A Propeller Chip program can consist of a main program using a single Cog with the other Cogs dedicated as low-latency interrupt handlers. For the Propeller Chip this allows up to seven low-latency interrupts. Because each Cog is running independently to the others all multiple interrupts will be responded to in parallel - something which no single-core microcontroller can achieve except when interrupts are handled by chip hardware.

The on-chip counter hardware each Cog has can be used for short signal pulse detection. These can be used by a Cog which is polling to detect a signalling event and then respond to it at its leisure.

A Cog which is polling multiple interrupts can implement its own prioritorisation scheme for interrupt handling. As interrupts are being handled by polling it is also possible to have interrupt events dependent on the state of multiple signal lines rather than just a single signal. The polarity of all signal lines monitored can of course be specified under programmer control and any of the available I/O lines can be used to trigger interrupt events. Prioritorisation and so on can all be changed dynamically at run-time if necessary.

A mix of Cogs can be allocated to handle low-latency interrupts and polled interrupts giving flexible options to implement what would be called an interrupt architecture on other microcontrollers. What type of interrupt architecture is required or implemented is entirely in the hands of the designer and programmer not forced by the chip designer.

Culture Shock

Users of more traditional microcontrollers are often taken aback when they discover that the Propeller does not support interrupts. For all intents and purposes it does, but not in the way that a single core processor has its program flow interrupted while an interrupt event is handled.

As with many things Propeller related (and it is the same with any multi-core processor) it is really just a case of doing things in different ways. What another microcontroller can do with interrupts can most likely be achieved by a Propeller Chip. That it "does not have interrupts" is really just a terminology and implementation issue.

What the Propeller Chip offers instead of traditional interrupts is far more flexible and equally as effective. "Not having interrupts" sounds like a major failing and problem, but in reality it is not.

I/O Bus Systems

Original discussion thread : here

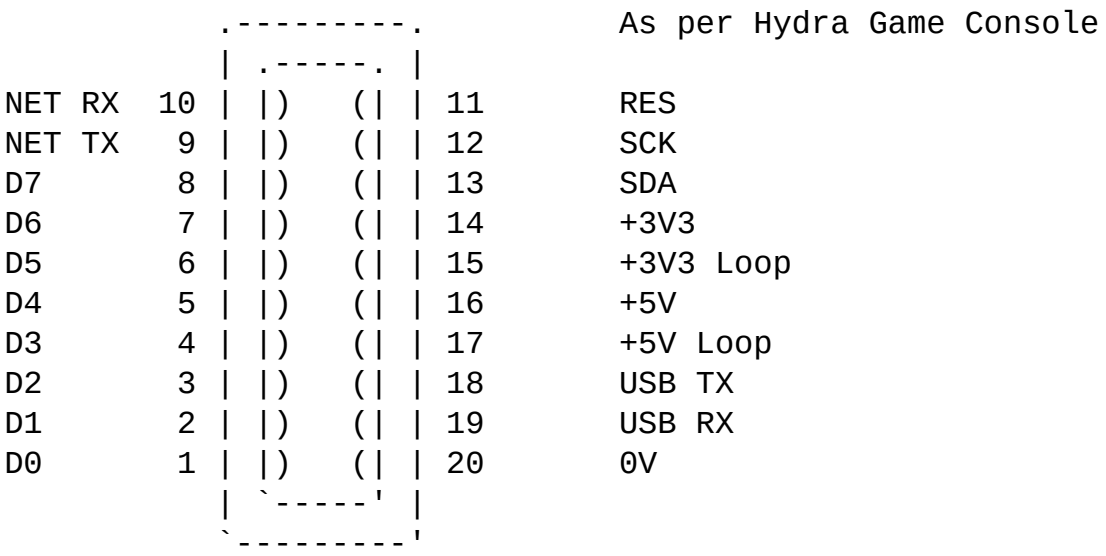
Existing I/O Bus Systems

Hybrid Development System
Hydra Game Station
MoBoStamp (MoBoProp)
PropBus
SpinStudio

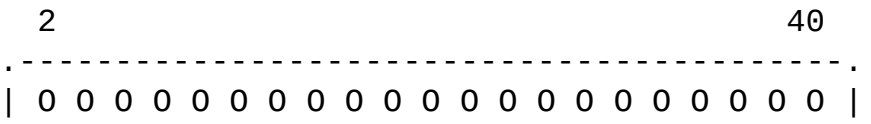
Hybrid Development System

Hybrid Development System Home Page : here

20-way, 10 x 2, 0.1" PCB Edge Connector, female socket on board



40-way, 20 x 2 IDC, male socket on board



```
        | O O O O O O O O O O O O O O O O O O O O |
        `--------------------^^^--------------------'
          1                                       39

      Connected one-to-one to 40-Pin DIP Propeller Chip
```

Hydra Game Console

Hydra Game Console Home Page : here

20-way, 10 x 2, 0.1" PCB Edge Connector, female socket on board

```
           .---------.
           | .-----. |
NET RX  10 | |)   (| | 11       RES
NET TX   9 | |)   (| | 12       SCK
D7       8 | |)   (| | 13       SDA
D6       7 | |)   (| | 14       +3V3
D5       6 | |)   (| | 15       +3V3 Loop
D4       5 | |)   (| | 16       +5V
D3       4 | |)   (| | 17       +5V Loop
D2       3 | |)   (| | 18       USB TX
D1       2 | |)   (| | 19       USB RX
D0       1 | |)   (| | 20       0V
           | `-----' |
           `---------'
```

MoBoStamp (MoBoProp)

Parallax Product Page : here

12-way, 6 x 2 IDC, male socket on board

```
 Daughterboard Connector (2mm)      DB-Expander SIP Adapter (2.54mm/0.
1")
           .---------.
Vdd      1 |  O   O  | 2    Vss          .-------------------------.
A7       3 `. O   O  | 4    A6           | O O O O O O O O O O O O |
A5       5  | O   O  | 6    A4           `-------------------------'
A3       7  | O   O  | 8    A2             V|A A A A A A A A|V|V|+
```

```
A1/An1    9 .' O   O  | 10   A0/An0          s|7 6 5 4 3 2 1 0|d|i|5
Vin      11 |  O   O  | 12   +5V             s|                 |d|n|V
            `---------'
```

PropBus

PropBus Home Page : here

10-way, 5 x 2 IDC, male socket on board

```
            .---------.
+3V3       1 |  O   O  | 2          0V
D0         3 `. O   O  | 4          D2
D2         5  | O   O  | 6          D3
D4         7 .' O   O  | 8          D5
D6         9 |  O   O  | 10         D7
            `---------'
```

SpinStudio

SpinStudio Home Page : here

20-way, 10 x 2 IDC, male socket on board

```
            .---------.
0V         1 |  O   O  | 2          0V
           3 |  O   O  | 4
D7         5 |  O   O  | 6          +5V
D6         7 `. O   O  | 8
D5         9  | O   O  | 10         +3V3
D4        11  | O   O  | 12
D3        13 .' O   O  | 14
D2        15 |  O   O  | 16
D1        17 |  O   O  | 18         SCL
D0        19 |  O   O  | 20         SDA
            `---------'
```

SpinStudio has four sockets "A" through "D", D0-D7 representing Propeller Pins P0-P7, P8-P15, P16-P23 and P24-P31 respectively.

D4 and D5 (the I2C lines) are not accessible on Socket "D" - they are pulled to VDD on the mainboard by 4.7K resistors and accessible only as I2C lines on pins 18 & 20 in all 4 Mainboard Sockets

You can chat with other Propeller users, in real time, using the Internet Relay Chat system. (IRC) All that is needed is a quick and free client download and the server and channel name, provided below. See you on #propeller @ irc.freenode.net!

Xchat for win32 can be downloaded here.
A web-based client connected directly to #propeller can be accessed from here.

Once you install it, run the software and it will display a list of servers and ask you for nickname choices. Choose freenode. It will then prompt you to choose a channel. Enter "propeller" after the pound sign, and you are connected!

We have an in-channel robot that can teach and learn. The following commands can be used by everyone to create and modify the fact database:

```
!learn [fact key] is [fact value]
```

The !learn command will create a list of facts with numbers if more than one value is associated with the same key.

The [fact key] can be multiple words but should be kept short. After learn, a fact can be queried directly:

```
!learn foo is A short variable placeholder.
propbot | username: The command succeeded.
!foo
propbot | username: foo is A short variable placeholder.
```

The [fact value] can be sentence form and may contain links. For particularly long links, please use http://tinyurl.com to shorten the url.

The database can be searched using the commands !q, !query, !k, !v.

!q, !query and !k are synonyms. They search the [fact key] portion of the fact database.

```
!k hub
propbot | username:  "hub operations" is See page 24 of the manual. Th
is thread has interesting q&a
          http://forums.parallax.com/forums/default.aspx?f=25&m=202526
```

The !v command searches values:

```
!v forums.parallax.com
propbot | username: 'indirect addressing', 'VGA Learning Driver', 'beg
inning assembly', 'gear',
                     'hub operations', and 'spin interpreter'
!gear
propbot | username: "gear" is http://forums.parallax.com/forums/defaul
t.aspx?f=25&m=242685
```

You can remove facts in order to edit them using the "!forget" command:

```
!forget foo
propbot | username: The command succeeded.
```

If there are multiple listings for the specified key, the !forget command takes a number argument, specifying which item to delete:

```
!foo
<propbot> username: "foo" is (#1) A short variable placeholder., or (#
2) The name of this factoid entry.
!forget foo 2
propbot | username: The command succeeded.
```

Large Memory Model

The Propeller comes with two in built programming environments:

- Assembler is the fastest possible code, but because it runs in Cog RAM it is limited to an absolute maximum of 496 instructions.
- Spin runs from Hub RAM, and so can use up to 32KB for code. But because it is a virtual machine running a byte code it is many times slower than assembler. Some say 40-100 times slower, depending on the specific code.

Large Memory Model (LMM) is an alternative programming environment suggested by Bill Henning. It lies somewhere between the concepts for native assembler and a virtual machine. It's a minimal virtual machine that runs on instructions that are mostly a 1:1 mapping with the native assembler instructions of the propeller. The instructions reside in Hub memory but are copied one by one into Cog RAM to be executed. This means that the code can amount to nearly 32KB of code, which is up to 8K instructions.

The basic virtual machine consists of just 4 lines of assembler:

```
nxt    rdlong  instr,pc
       add     pc,#4
instr  nop                ' placeholder!
       jmp     nxt
```

As can be seen, at ***nxt*** an instruction pointed to by the Program Counter (pc) is copied to Cog memory at ***instr*** (replacing the ***nop***). Then after incrementing the program counter to the next long, the instruction is executed. This loop takes 32 cycles, and so is 8 times slower than native assembler for executing code, but several times faster than spin. However the loop can be unrolled to make it faster - Bill suggests unrolling 4 times to get it just 5 times slower than native assembler.

Additionally note that the instruction executed could call a routine that lies elsewhere in Cog RAM. This technique can be used to create extra pseudo instructions that don't exist in the native assembler set. Indeed some are needed for jumping and calling to LMM code elsewhere in Hub Memory. Bill suggests these pseudo instructions:

```
* FJMP addr        - calls routine that replaces PC with long at PC, th
en jumps to nxt
* FCALL addr       - increments SP by 2, replaces PC with long at PC af
ter it saves
                    PC+4 at SP
* FRET             - loads PC from word at SP, decrements SP by two
* FBRC addr        - branch to far address if Carry flag is set
* FBRNC addr       - branch to far address if Carry flag is clear
```

```
* FBRZ addr       - branch to far address if Zero flag is set
* FBRNZ addr      - branch to far address if Zero flag is NOT set
* FCACHE [code] 0 - copies a block of code, terminated by NULL into a
cache area in
                    Cog memory. Then executes it.
```

Implementations

At the time of writing, a few people have experimented with the scheme and written some small LMM programs which do work. However there is as yet no definitive version of the scheme, and various tools implement slightly different versions of LMM code.
Bill Henning is working on an LMM Macro Assembler/Linker.
ImageCraft have implemented LMM in their C compiler.
Ross Higson has implemented LMM in his Catalina C compiler
More details and much discussion can be found in the original thread (old forum).

LMM Kernel Specification - Pacito version

Ale500 (aka Pacito) has drafted a Specification for a LMM usable even beyond the 32 kbytes imposed by the HUB RAM using an external means of accessing more storage.

The specification for a Large Memory Model with access up to 512 klongs of code and 512kbytes of data can be found here.

LMM Kernel Specification - AiChip Industries version

AiChip Industries have created a Large Memory Model Virtual Machine implementation (LmmVm) which is designed to be usable with the Propeller Tool. Some native Propeller instructions need to be modified for LMM usage but those changes are designed to be achievable by hand rather than requiring any additional tools.

Details of the Large Memory Model with access up to 32klongs of code or data and using cog-based registers can be found here.

LMM Kernel Specification - Phil Pilgrim (PhiPi) version

Phil Pilgrim (PhiPi) examined how the main LMM loop is usually written and determined a way to overcome that loop's failure to hit 'hub access sweet spots' which requires the LMM loop to be un-rolled to achieve maximum throughput and lowest speed degredation when compared to native PASM

execution.

The mechanism used rests upon reversing the LMM code so addresses of LMM Code reduce sequentially rather than increment and use a clever sequence of PASM instructions to maximise LMM throughput.

Details of the Reversed Large Memory Model can be found here.

Thumb-style Code

An extension to the Large Memory Model scheme is a Thumb-style (similar to that developed by ARM(R)) representation where 16-bit (word) codes are used to represent a subset of the native 32-bit Propeller instructions. Word instructions are fetched from hub memory, decoded, expanded and then executed.
While consequently slower in execution than the Large Memory Model, up to 16K instructions can be held in hub memory. Execution should also be faster than when interpreting any arbitrary bytecode such as that used by Spin.

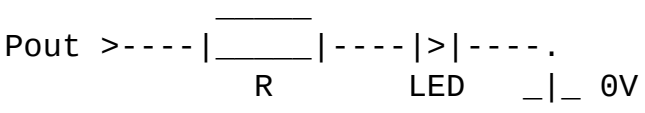
AiChip Industries has proposed a Thumb VM implementation and produced proof of concept code. Without appropriate development tools the use of Thumb VM is not practical at the current time.

AiChip Industries' Thumb VM Proposal can be found here.

The original discussion thread and proof of concept can be found here.

Active High LED

The LED will be illuminated when the Pout pin is made an output and is taken high.

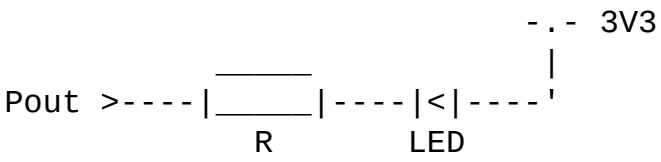


Example Spin program for a LED on pin 0 ...

```
DIRA[0] := 1    ' Make pin 0 an Output
OUTA[0] := 1    ' Make Pin 0 high, LED on
OUTA[0] := 0    ' Make Pin 0 low,  LED off
```

Active Low LED

The LED will be illuminated when the Pout pin is made an output and is taken low.



Example Spin program for a LED on pin 0 ...

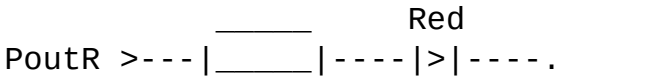
```
DIRA[0] := 1    ' Make pin 0 an Output
OUTA[0] := 0    ' Make Pin 0 low,  LED on
OUTA[0] := 1    ' Make Pin 0 high, LED off
```

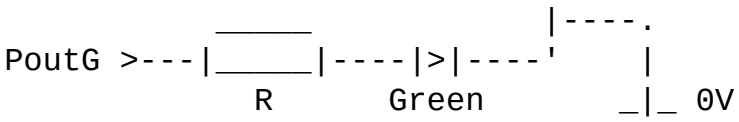
Tri-Colour LED

The Red LED will be illuminated when the PoutR pin is made an output and is taken high.

The Green LED will be illuminated when the PoutG pin is made an output and is taken high.

Both LED's will be illuminated (a yellowish colour) when both the PoutR and PoutG lines are made outputs and taken high.





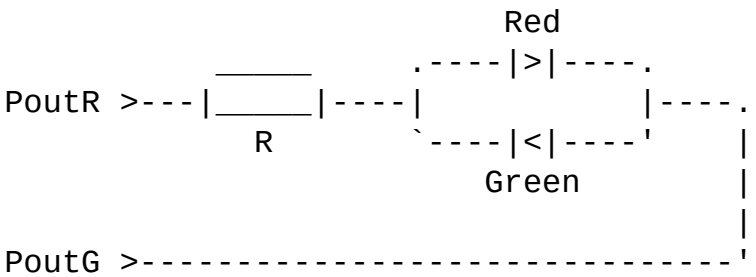
Example Spin program for a LED on pin 0 (Red) and pin 1 (Green) ...

```
DIRA[0] := 1    ' Make pin 0 an Output
DIRA[1] := 1    ' Make pin 1 an Output

OUTA[0] := 1    ' Make Pin 0 high, and ...
OUTA[1] := 0    ' Make Pin 1 low,  Red LED on

OUTA[0] := 0    ' Make Pin 0 low,  and ...
OUTA[1] := 1    ' Make Pin 1 high, Green LED on

OUTA[0] := 1    ' Make Pin 0 high, and ...
OUTA[1] := 1    ' Make Pin 1 high, both LEDs on

OUTA[0] := 0    ' Make Pin 0 low,  and ...
OUTA[1] := 0    ' Make Pin 1 low,  both LEDs off
```

Bi-Colour LED

The Red LED will be illuminated when the PoutR pin is made an output and is taken high while the PoutG pin is made an output and is taken low.

The Green LED will be illuminated when the PoutG pin is made an output and is taken high while the PoutR pin is made an output and is taken low.



Example Spin program for a LED on pin 0 (Red) and pin 1 (Green) ...

```
DIRA[0] := 1    ' Make pin 0 an Output
DIRA[1] := 1    ' Make pin 1 an Output

OUTA[0] := 1    ' Make Pin 0 high, and ...
```

```
OUTA[1] := 0    ' Make Pin 1 low,  Red LED on

OUTA[0] := 0    ' Make Pin 0 low,  and ...
OUTA[1] := 1    ' Make Pin 1 high, Green LED on

OUTA[0] := 0    ' Make Pin 0 low,  and ...
OUTA[1] := 0    ' Make Pin 1 low,  both LEDs off

OUTA[0] := 1    ' Make Pin 0 high, and ...
OUTA[1] := 1    ' Make Pin 1 high, both LEDs off
```

Propeller Links

A list of useful and interesting sites related to the Propeller (in Alphabetical order):
HSS Sound Driver - Hydra Sound System
Ignite Automation - PropBus modular development system
ImageCraft - Makers of a C compiler for the Propeller (ICCv7)
Object Exchange - Repository for various Propeller software objects of use by other developers
Parallax - Creators of the Propeller
PropDOS - A miniDOS for a Propeller with SD interface
Propgfx - Graphics card using a Propeller
Proptcp - TCP/IP Stack for the Parallax Propeller Chip
SpinStudio - A plug together construction set for Propeller electronics
TinyMicros wiki - assorted tips and tricks for the Propeller chip
TOMIG - The Orrville Microcontroller Interest Group
Wulfenden.org - Propeller Robot Controller
XGameStation - André LaMothe's Hydra Propeller based game console

This is a mirror of Spork Frog's excellent instructions, originally published on the Propeller Forum A new set up updated instructions can be found here

Developing with the Parallax Propeller Microcontroller under Linux-Based Operating Systems

A step-by-step guide
Rev. 0B

This is a guide to help you set up a development environment for the Propeller under native Linux without the use of VMWare, Bochs, QEmu, VirtualBox, or other similar virtualization products.

Step 1: Prerequisites

All of the following should be installed prior to any of the other steps. Most popular distributions provide these; see your package manager for more information.

- python
- pyserial
- wine

Step 2: The Propeller Loader

The one thing when dealing with the Propeller that never seems to work correctly is communication with the chip itself. However, thanks to the great work of Remy Blank, there is now a Python script that runs natively to handle communication. Although it's not as fast as the Propeller IDE, it still gets the job done just fine.

You can find the script here.

Step 3: Propellent

Parallax has recently released a command line version of their popular Propeller IDE and compiler. It can take in a SPIN file, compile it, and with your choice either save a binary/EEPROM file or upload to the Prop directly. Direct upload seems once again not to work under Wine, but the compiler works fine.

You can find it on Parallax's website here.

Step 4: Tying it all together

Copy the following files to the same folder as the SPIN file you wish to compile:

```
Loader.py
Propellent.dll
Propellent.exe
```

Edit your SPIN code in your favorite text editor.

Then, through a command line, run the following:

```
wine Propellent.exe /compile yourfile.spin /savebinary
python Loader.py yourfile.binary -r
```

LMM Kernel Specification - AiChip Industries version

AiChip Industries have created a Large Memory Model Virtual Machine implementation (LmmVm) which is designed to be usable with the Propeller Tool. Some native Propeller instructions need to be modified for LMM usage but those changes are designed to be achievable by hand rather than requiring any additional tools.

The LmmVm allows to up to 8k longs of code or data and allows access to cog-based registers for fast run-time data storage.

The LmmVm is is a linearly executing LMM engine which takes one instruction at a time, executes it, and moves on to the next. There is no caching, overlays or 'block load and execute' operations supported. The LmmVm is therefore most efficient with sequential rather than looping LMM code, however the overhead of looping will often be tolerable for many LMM programs.

The LmmVm executes most native Propeller Assembler instructions in LMM mode without change (including conditional execution plus WC, WZ and NR options) but branches, subroutines and some data access instructions need to be replaced by calls into the LmmVm itself to perform as required. These calls into the LmmVm are followed by parameters which indicate what the call is to achieve and are usually specified by 'long' constant settings. This is less eficient in code space than it could be but allows the calls to be reasonably easily created by hand.

Instructions which need to be replaced by calls into LmmVm are -

jmp
call, ret
djnz, tjz, tjnz

In addition, the LmmVm provides support for conditional execution of calls into the LmmVm (Conditional , Skip) and for operations which would normally be done using self-modifying code with native Propeller instructions (Get Register Indirect, Store Register Indirect, Execute Register, Case Jump). The ability to load a register with a 10-bit to 32-bit constant which would normally require the allocation of pre-loaded Cog register is also added (Load).

To overcome the need to modify native Propeller 'ret' instructions on subroutine calls, the LmmVm implements a a software stack which can either be held in Cog memory or in Hub memory. Benchmarking shows there is little, if at all any, impact on performance of LmmVm regardless of which is used.

LmmVm is designed to extensible. New LMM instructions can be freely added to the LmmVm as required. The only consequence is that additions will reduce the Cog memory available for LMM program register and stack use. Conversely, LMM support can be reduced where such specific support is not required.

LmmVm can be run standalone, executing programs written entirely as LMM programs, but it may also be used as a sub-component of any other VM which needs to perform tasks where Cog memory

limitations mean that not all required code can be held within the Cog memory. The 'Primary VM' will pass control to the LmmVm indicating where the LMM code is to execute, and the LMM program upon completion will jump back to the Primary VM. In this way a Primary VM can have full-speed assembler execution when required and use slower speed LMM execution as necessary. This is convenient when designing any Primary VM as the Cog memory limitations can be ignored during development. Speed critical parts can be brought back to within the Cog memory for native execution as development nears its end. LmmVm evolved from originally being a sub-system in such a Primary VM.

LmmVm allows Primary VM's to be written as LMM code which can execute end-user program code (bytecode) which has better code density than that provided for by Propeller Assembler or LMM code. The LMM program interprets the bytecode program, while LmmVm interprets the LMM code. Again, Cog memory limitations can largely be ignored while the Primary VM is being developed (using LMM code) although each level of interpretation does result in a lower overall execution speed of the interpreted bytecode.

As well as providing a VM support service for Primary VM's, LmmVm can also be utilised by any Cog program which needs to execute more code than can be held in Cog memory and is a general purpose tool for any Propeller Assembler Programmer.

Specification for a Large Memory Model with access up to 512 klongs of code and 512kbytes of data.

LMM Kernel Specification v1.0 - pacito version

The purpose of this specification is to serve as basis for a large memory model (LMM) for the Parallax Propeller. This would extend the usable memory area beyond the 2kbytes of COG's memory. This specification requiers the following hardware support:

- A parallax propeller v 1.0
- Am external RAM (DRAM or SRAM) for more than 32 kbytes of code

Note: Instruction names refer to the COG native instruction set. When a reference to a new instruction is made, its functionality is implied. The program refers to the un-preprocessed source with a mixture of native and no native instructions. The compiled (or compiled program) refers to the above mention program after preprocessing and compilation. It only contains COG machine code.

Scope

The COG can execute very fast machine code, so It makes sense to use this built-in language instead of creating a second level language. If some instructions are worked around limitations to memory access can be bypassed.

- Memory addressing is limited to 512 memory locations
- Program execution is available beyond 511 memory locations
- No support for high-level languages like C
- Up to 512 registers

The last point, as the cog works is a strength, but in our case as the program is going to be relocated has to be reduced to a mere 16 registers, with 8 of them having special meanings.

R0..R7: general purpose registers
PC : program counter
SP : Stack pointer
DP : Data pointer (points to the beginning of the Data section)
IM : Immediate register, used by the kernel
BP : Base pointer, used for stack access
DL : Data length, length of the Data area
U0, U1 : not yet defined

To address the first two points a new set of instructions is proposed:

ldb, ldw and ldd : These instructions can read any memory location, at byte, word and long lengths. Their counterparts stb, stw and std write to any memory location.

Jump, subrutine call and return are also supported via rcall and rjmp. These instructions can access any memory position inside program memory. Stack is handled automatically.

Stack manipulation, push, pop, and stack read and write in place with a base pointer is also provided via special instrucions ldbpb, ldbpw, ldbpd, stbpb, stbpw and stbpd. As part of the stack manipulation, two more instructions enter and leave provide stack reserving techniques. Stack overflow is checked for every new enter instruction.

A new instruction to load long immediates from program memory is provided, too.

Some of these instructions will be subroutine calls to kernel functions, and some will require arguments. Extra arguments will be either a long constant after the instruction (jumps and loads/stores from memory) or a mov instruction before the call to a special kernel register in the case of small arguments. Thise will create multi-long pseudo instructions. As the program will be divided in 32 longs chunks, special care is taken to not have a double long instruction on the last long of a chunk. A nop will be added in these case.

Memory management

As memory inside the cog is limited to 496 instructions, the only way to extend it is to use caching techniques. The cog's memory will act as a Level 1 cache and the HUB memory can act as a secondary cache if desired when external memory is used, in the case that only HUB memory is used, this feature can be left out.

The cache will occupy 128 longs and will have 4 chunks (lines) of 32 longs each, leaving the rest to the kernel. Caching occurs transparently to the application and with special support from the extra instructions use of the increased address space can be done. These lines will be filled on-demand and reused as needed (without overwriting the last used line in a round-robin fashion). An aging mechanism could be implemented if space allows.

The cache line contains two extra fields, a memory pointer to its absolute address (aligned to a 32 long boundary) and a jump instruction at the end to return to the kernel. The first allows for fast look-up in case of a rjump / rcall instruction and the second return the control to the kernel to load another cache line or to continue execution if the line is already present. No multilong instructions are allowed to be separated in two cache lines, the preprocessor will ensure this.

Data memory and stack are accessed outside the scheme. Self-modifying code is not possible at this time.

Control transfer instructions

The program shall have only rcall, rret and rjmp instructions, these instructions support transfer of control to the whole memory space beyond cogs's memory limitation using the techniques described in the caching section.

Each new instruction will be replaced by the preprocessor for a sequence of native cog machine code.

rcall #a_function_beyond_2k

is going to be replaced by

call #krnl_rcall
long a_function_beyond_2k

The long is going to be ignored by the COG because its condition bits are zero. That will limit the addressable area to 19 bits, that would mean 2^19 longs, but the same restriction exists for memory load/store, so it is limited to 2^19 bytes.

rret is going to be replaced just with a call to krnl_rret. No extra arguments
are needed.

The case of jumps is a bit more complicated. If jmp were to be used, the line would have to be relocated appropriately, for that purpose all jumps will have to be rjumps. Using the scheme before, conditional jumping can be used without problems, because the second parameter has condition IF_NEVER (all zero), the call/jump can have a condition. That will somehow reduce the overhead.

A special rjmp could be used if the jump would occur inside the cache line, but memory constraints, i.e. no place to implement it, can limit its availability.

Memory load/store

The special instructions ldb, ldw, ldd, stb, stw and std are used to access data in the data section. They are plus the DP used to address up to 512 kbytes of data.Read and write to memory occurs without caching.

Stack load/store

The stack has its own group of instructions for stack manipulations. These lay also in a 512 kbyte area.

Program termination

The program terminates when the instruction term is executed, this instruction is replaced by a call to the kernel routine krnl_term.

Syscall and interaction with the hardware

The syscall mechanism is going to be handled over to another COG, for that purpose, an area of HUB memory will be used. This points need working.

Additional hardware

To use more than 32 kbytes of code/data, an external memory is needed. Two possible connection methods have been envisioned, (but more exist). A SRAM or DRAM connected directly to the Propeller executing the LMM kernel, or a SRAM/DRAM connected to a helping circuit, a CPLD or a second

propeller. This second approach is being tested as of now. This helping circuit is responsable for obtaining the data to fill the cache lines at a rate of one long every 5 or 6 instructions, generate the lower addresses and to interface with the memory device.

- End of the proposed specification ***

LMM Kernel Specification - Phil Pilgrim (PhiPi) version

Phil Pilgrim (PhiPi) examined how the main LMM loop is usually written and determined a way to overcome that loop's failure to hit 'hub access sweet spots' which requires the LMM loop to be un-rolled to achieve maximum throughput and lowest speed degredation when compared to native PASM execution.

The mechanism used rests upon reversing the LMM code so addresses of LMM Code reduce sequentially rather than increment and use a clever sequence of PASM instructions to maximise LMM throughput.

The original discussion thread on this Reversed LMM concept can be found here

The main LMM loop of most LMM interpreters had previously used the following code -

```
              mov     pc,PAR

Loop          rdlong  Instr,pc
              add     pc,#4
Instr         nop
              jmp     #Loop
```

The LMM Loop for the Reversed LMM interpreter is -

```
              mov       pc,PAR
              jmp       #Start

Instr1        nop
              rdlong    Instr2,pc
              sub       pc,#7
Instr2        nop
Start         rdlong    Instr1,pc
              djnz      pc,#Instr1
```

LONG

In Spin a long is a signed integer. Unlike words and bytes which are unsigned. In assembler it's your choice whether it's signed or not, depending on what operand variants you use.

LONG is used as a keyword in 4 different ways:

- In a VAR block
  - `LONG Symbol`
- In a DAT block
  - `LONG ...`
  - `LONG [ count ]`
- In a method
  - `LONG [BaseAddressInBytes]`
  - `LONG [BaseAddressInBytes][OffsetInLongs]`
- In a method
  - `Symbol.LONG[OffsetInLongs]`

LONG Symbol

Declaration of a Spin long variable. Guaranteed to be long aligned. When compiling, Spin groups all the long declarations together in a block before all the word and byte declarations, so you can't count on the order of differently sized variables in memory being as in the source. However, all same sized variables will be in the order you declare them.
These variables only exist in Hub memory. They will exist at a place past the binary image combined by PropTool.
They are always initialised to zero.
To access them from assembler, you'd have to pass the address of one to the assembly program through the PAR mechanism and use **RDLONG/WRLONG**.

LONG ...

LONG [count]

Declare a long aligned label. Layout in memory will reflect the order declared in the source, however differently aligned declarations may result in padding.
The data exists in Hub RAM, and may be copied to Cog RAM when starting a Cog. Spin references will use the original in Hub RAM, Assembler references will use the Cog RAM copy (unless done by reference though **PAR** and **RDLONG/WRLONG**).
When a long data value is followed by a count in square brackets, that number of longs will be created. This is useful for pre-initialising block or arrays of data within a Cog.

LONG [BaseAddressInBytes]

LONG [BaseAddressInBytes] [OffsetInLongs]

In spin will read/write to a word in Hub RAM. It can only do long aligned read/write, in other words it ignores the least significant two bits of BaseAddressInBytes.

```
{{LONG [BaseAddressInBytes] := value}}
'Is equivalent to:
{{BYTE[ BaseAddressInBytes & $FFFE ] := value & $FF}}
{{BYTE[ BaseAddressInBytes | $0001 ] := (value >> 8) & $FF}}

{{LONG [BaseAddressInBytes] [OffsetInLongs] := value}}
'Is equivalent to:
{{BYTE[(BaseAddressInBytes&$FFFE)+(OffsetInWords*2)] := value & $FF}}
{{BYTE[(BaseAddressInBytes|$0001)+(OffsetInWords*2)] := (value >> 8) &
 $FF}}
```

Symbol.LONG[OffsetInWords]

In spin will read/write to a long in Hub RAM. Symbol must be a long variable. It'd more straightforward to use simple array indexing - **`Symbol[Offset]`**.

See also

WORD
BYTE
LONG vs RES
Symbol Address operator

How is RES different from LONG?

This is very difficult to explain, but understanding this is one of those Eureka moments that make you into a real Propeller assembly programmer.

As PropTool is compiling a spin program, with embedded assembler, it is keeping track of two important pointers.

1. The Hub RAM pointer. This is the address in Hub RAM that the current code or data is going to be placed. It starts at zero, and keeps getting incremented as spin code is compiled or assembler is assembled. It measures memory as bytes.
2. The Cog RAM pointer. This is set by the ORG directive, and is then incremented as assembler instructions and LONG, WORD and BYTE statements are assembled. It represents the eventual Cog address where code and data will live once it have been copied into a cog by a COGNEW or COGINIT. It measures memory as longs.

LONG defines an item of data. It increments the Cog RAM pointer by 1 and the Hub RAM pointer by 4.
RES **res**erves what will eventually be a space in Cog RAM with undefined content. It just increments the Cog RAM pointer by 1. It leaves the Hub RAM pointer alone.
That is the difference.

Why?

When you have a long that has initialised data, then it must occupy a long in Hub RAM to start with, and then it is copied with the rest of the code to a Cog to be executed. If it doesn't need to be initialised, you could just store any old value (say zero) in the long in Hub RAM and have that copied. But this unnecessarily wastes a long in Hub RAM that isn't needed. RES is a statement that allows you to mark out space with labels at the end of Cog RAM, without that space having to be taken up in Hub RAM.

"I don't get it. Just tell me what I need to do"

- If you need initialised data in your assembler routine, e.g. from constants >$1FF, then use LONG to define them.
- If you don't need data to be initialised, then use RES. But ensure that for a particular block of code and data (between ORG and FIT) that all RES lines comes after everything else.
- If you really don't understand what all this means, just always use LONG and leave RES alone.

Another explanation

Here's what Propeller Tricks and Traps has to say on this issue:

Be sure to place RES statements at the end of any ORG segment that uses them. The following example does it wrong:

```
                //MOV     Time,CTR
                ADD     Time,Offset
                ...

Time            RES     1
Offset          LONG    230
Other_value     LONG    123

                ORG
Another_cog     ...//
```

In this example, Offset will get clobbered when CTR is copied to Time, and 123 will get added to it instead of 230. Why? Because when the assembler encounters a RES, it reserves space in cog memory, but not in hub memory where the program is stored. Consequently, 230 will occupy Time's address in hub memory, and 123 will occupy Offset's. Do this instead:

```
                //MOV     Time,CTR
                ADD     Time,Offset
                ...

Offset          LONG    230
Other_value     LONG    123
Time            RES     1

                ORG
Another_cog     ...//
```

In this example, whatever's assembled at Another_cog will get loaded into Time when the first cog loads. But we don't care, since Time will get written over anyway.

BST: Brad Spin Tool

The bst tool suite is a cross-platform tool set to help non-windows users get the most out of their Propellers. BST also runs on Windows.

Binaries are provided for
i386 Linux
i386 Windows 95->XP
Universal Mac Binaries for OSX 10.4, 10.5 & 10.6

The Linux command line binaries are completely statically linked and will run without issue on x86_64 systems.
The Linux binaries have been tested down to 2.4 kernels, so pretty much anything made in the last 6-7 years will work fine.
If it doesn't, then let me know and I'll fix it!

I'm particular about supporting MacOS Intel & PowerPC and Linux. Windows bugs and features are welcome also.
Just an update. The bst suite now has a homepage (such as it is)
http://www.fnarfbargle.com/bst.html

Download it here : http://www.fnarfbargle.com/bst after reading the bits below!

The suite consists of :

bstl : The propeller loader

This little application does nothing more than allow you to load pre-compiled .binary and .eeprom files into your propeller.
It is a command line application that takes a couple of optional parameters and a file name. Nothing more, nothing less.
Use bstl -h to get a list of what does what.

bstc : The Spin Compiler

bstc has bstl built in, in addition to a completely Parallax compatible SPIN and PASM compiler and linker.
It has a few extra features (like being able to emit a list file, some basic optimisation, and zip file generation)
As with bstl, use bstc -h for a list of what does what.

bst : The IDE

This is the top of the food chain. A complete windowed IDE that aims (and still falls short) to be comparable with the Parallax Propeller Tool.
It's simply an environment for developing your propeller code, downloading and monitoring the results.

bst is structured to be a work-alike to the Parallax Propeller Tool so that new users will find the Parallax documentation familiar enough to be able to start from scratch without being plunged into an unfamiliar environment.

The biggest hurdle in getting bst functioning up to scratch is the installation of a suitable Font. A modified version of the Parallax Propeller Font can be found linked from this Wiki here :
http://propeller.wikispaces.com/Propeller+Font
Please install this font prior to running bst the first time and it will ensure the smoothest installation.

bst has a basic serial terminal built in. It's not fantastic but it does the job. Suggestions warmly welcomed. Copy and paste as well as saving the contents is supported.

The development of bstl, bstc and bst can be followed on the Parallax forums linked below, and release notes and Changelogs are often posted in the top post.

bstl : http://forums.parallax.com/forums/default.aspx?f=25&m=297274
bstc : http://forums.parallax.com/forums/default.aspx?f=25&m=297566
bst : http://forums.parallax.com/forums/default.aspx?f=25&m=298620

Extensions to the Official Spin Compiler in PropTool

A number of extensions have been added. In BST they can be enabled using the Optimisation Panel in the Tools->Compiler Preferences Item. This extensions are not compatible with PropTool that means that the code generated is not going to be byte by byte the same as the one produced by PropTool (whereas when they are not enable bst strives to generate bit for bit identical code).

One of the most interesting extensions is **conditional compilation**. It works is a similar manner to C's, using the same syntax. Text replacement and macro expansion are not yet supported (v 0.18.4).

The syntax is similar to homespun's:

```
* #define symbol
* #undef symbol
* #ifdef symbol
* #ifndef symbol
* #elseifdef symbol
```

```
    * #elseifndef symbol
    * #else
    * #endif
    * #info I'm an information message
    * #warn I'm a warning message
    * #error I'm an error message and will abort compilation
```

bst[c] passes the defines down to sub-objects as they compile. You will get an error if you try to define a symbol twice.

#ifdef and #elseifdef check to see if symbol has been defined using #define (see below).
Example:

```
#define HYDRA

#ifdef HYDRA
  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000
#elseifdef HYBRID
  _clkmode = xtal1 + pll16x
  _xinfreq = 6_000_000
#else
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000
#endif
```

Another interesting extension is the @@@ operator. It returns the HUB address of the specified symbol.

Any comments, questions and suggestions are more than welcome to : proptools at fnarfbargle -dot- com
Note: Do not edit the wiki to post questions, just email me at the above address or use the parallax forum Thread.

Max OS X Experiences

The intent of this page is to assist those using the various Propeller tools on the Apple Mac OS-X platform.

Native tools

The BST tool is a third party clone of Spin Tool, which runs on OS X (in addition to Windows and Linux).

PC Emulation using Parallels

It is possible to use all of the Propeller tools with Mac OS-x Tiger and Leopard utilizing the Parallels V3 software. At first there were some minor problems with USB ports but that has been fixed by Parallels. You can share files, software and clipboards.
Mac How

Boot Camp

Unknown

Software running on Multiple COGs is equivalent to multiple threads in a single process, and the same sorts of concurrency protection is needed to ensure correct operation. Race conditions are very easy to create. A typical design pattern is a object that uses a number of spin utility methods to access values stored in hub ram (DAT or VAR), and a loop in Spin or PASM main loop that runs on a COG. This generally limits the interaction of the utility methods and the main loop is by the Hub RAM.

Access to HUB ram is atomic for 32 bit reads OR writes. You cannot read and then write back a modified variable atomically. You cannot read more than one long variable atomically. The speed of the code doing the read and writes is likely to be very different as at least one side will be interpreted spin code, and the speed of spin operations is not transperant.

To ensure correct/sane results when pulling data from a Object that uses a COG thread needs to use some way to avoid race conditions. Techniques applicable for Propellers are:

- use a lock
- read a single 'output' variable. i.e. calculate interesting values in the cog and store in a hub ram.
- pack/unpack multiple variables into one LONG (i.e. atomic changes)
- read AND write groups of variables in very carefully timed code so that a race can not occur.
- start and stop the COG on demand (seems probably fast IF it is spin)
- Use some form of queue and/or flag in hub ram for handshaking.

If the value read or written by the utilities is independent, then no further precautions are needed. A variation would be to pack a group of variables into a long.

Handshakes via hub ram can be very simple and effective, so long as the handshakes are one direction. If only dealing with 2 variables, it is pretty simple to ensure a race does not cause in correct behavior.

By carefully analysing the order of memory access and modifications, races may be able to be avoided. This works because the COG's hub access is in lock step, so if one process copies in changes at the same rate as another is reading them out, they cannot overlap and get mixed values. However this would be rather difficult to maintain, as it requires analysis of the spin and/or PASM.

Starting and stopping a COG running spin is actually very fast, and in some cases it may turn out to be practical to stop the cog, and then restart it again. However to be safe in this approach, the stopping of the COG will probably need a handshake to avoid corrupting the state due to partially completed variable updates.

Another related problem is that the looping COG might never wake up if it is using a WAIT type instruction for an external event to occur. Stopping the COG maybe possbible if there is a way to avoid corruption of object variables (lock, handshake, or limited preserved state) Another alternative is to use a IO pin as a interupt and add it to the wait mask.

Example: Calculating speed

Calculating speed from a tacho type source will normally involve 2 values for the time references. From

this the worker COG running the main loop can calculate the speed value from its internal variables, and store it in a hub location, after each pulse. But what do you do when it has stopped and there is NO pulses?

In this case it is probably best is to store the BOTH the speed AND then the time of the last pulse, in that order. Check for a RECENT pulse BEFORE using the speed. In this case, a trick is that you need to also watch for roll over if you are using CNT as the time reference. In that case avoid the risk of a false pulse of speed every rollover, WRITE back a speed of zero when stopped is detected. This leaves a small risk of a speed Zero being reported until the COG queriing it has completed antother poll cycle, which is effectively no consequence.

Stopped Cogs

Some designs might be a bit simpler with a interupt to trigger the COG to do something that is outside its normal operation. An example could do with a timer interupt, OR they have to use locking

This page is still under construction, please use the discussion page for changes/improvement, comments, etc (Ale)

MATH

The propeller contains a 32 bit ALU with support for signed and unsigned basic arithmetic, a barrel shifter that makes all the difference and logic operations. Here it is assumed that you are familiar with the mnemonics used by the assembler, and that you know a bit of math. Some of these topics were covered in the "Propeller guts" document.
(Note: all code was tested using pPropellerSim and in the case of BCD math an actual propeller, too!), but bugs could exist, use at your own risk, and read the terms of the license.

Integer math, the four operations

Addition and subtraction are straightforward, multiplication and division require a bit more work.

```
add    x,y    wc, wz
```

Assuming **x** and **y** contain already the values to add and those are of the unsigned type. The **C** flag will signal overflow, i.e. the result is bigger than 2^{32}-1, and the **Z** flag will signal a zero result.

For signed numbers **adds** perform the same operation. A signed number has a smaller range from 0 to 2^{31}-1 on the positive side and -1 to -2^{31} on the negative side. So 1 + -1 will be zero (really?) and -3 + -4 will be -7 (unbelievable). As the range is smaller than unsigned, $7FFF_FFFF plus $7FFF_FFFF will give $7FFF_FFFE and will rise the **C** flag.

```
adds    x,y    wc, wz
```

The **C** flag will signal again overflow if the result exceeds 2^{31}-1 or -2^{31}. The **Z** flag will signal again a zero result.

If you think that 32 bits is not enough you can concatenate several operations together using the **C** flag to extend the word size, so for 64 bit arguments we have (**addx** is the addition with carry version of **add**).

```
add    xl,yl    wc
addx   xh,yh    wc, wz
```

The **addx** instruction will use the carry from the first operation, on the lower 32 bits of the number, if any

and add it to the upper 32 bits. **C** and **Z** work as before. If you need still more precision, more **addx** instructions can be chained as seen before.

Subtraction works in a similar manner using **sub** and **subx**:

```
                sub   xl,yl   wc
                subx  xh,yh   wc, wz
```

To multiply the propeller uses the good old addition and shift method due to the lack of a multiply instruction. But before that let us consider some special cases: multiplication by constants. As we know constants have the ability to conserve their value over time (!), so a fixed multiplication can save a few longs here and there. The propeller has a barrel-shifter that is essential for this to be smaller thatn using the normal multiplication depicted below. As we also know multiplication can be distributed across addition and that is the key to many common values:

For x*10 = x*2+x*8 = (x+x*4)*2

```
                shl   x,#1
                mov   r,x
                shl   x,#2
                add   r,x
or...
                mov   r,x
                add   r,x
                shl   x,#3
                add   r,x
```

For x*80 = x*16+x*64

```
                shl   x,#4
                mov   r,x
                shl   x,#2
                add   r,x
```

The source argument is **x** and **r** is the result. The propeller lacks a **lea** (load effective address) instruction so some neat tricks that can be exploited in x86 or 68k assembly, like multiplying by 3, 5 and 9 in one instruction, are out of the question.

The good old shift-add method works with two variables and a temporal register for counting (usable up to 16*16 bits):

```
                mov   r,#0
loop
                shr   y,#1   wc
        if_c    add   r,x
                shl   x,#1
                tjnz  y,#loop
```

This will only work if the lower part of a 17*16 to 32*32 bits is desired. Detection of overflow (**r>**2^{32}) requires that the overflow of the **add** instruction be honored and passed through, of course after the **tjnz** the status of the **C** flag must be checked.

```
                mov   r,#0
loop
                shr   y,#1   wc
        if_c    add   r,x    wc
                shl   x,#1
        if_nc   tjnz  y,#loop
```

If a full 64 bits result is needed (**rh:rl**)... well some changes are required:

```
                mov   rh,#0
                mov   rl,#0
                mov   xh,#0
loop
                shr   y,#1    wc
        if_nc   jmp   #loop2
                add   rl,xl   wc
                addx  rh,xh
loop2
                shl   xl,#1   wc
                rcl   xh,#1
                tjnz  y,#loop
```

Of course there are variations, this can be unrolled if we know how many effective bits one of the arguments has. If **y** is always smaller than **x**, it is better to test **y** against zero (**tjnz** instruction) than to test **x**. This can reduce the running time. These examples were coded for unsigned numbers, in the case of signed ones, the instruction **abs**, previous sign test could be used to produce the right result:

```
                mov   s,x     ' saves sign of x
                xor   s,y     ' calculates sign of result
                abs   x,x     ' calculates absolute value of x
                abs   y,y     ' of y ...
                mov   r,#0
loop
                shr   y,#1   wc
        if_c    add   r,x    wc
                shl   x,#1
        if_nc   tjnz  y,#loop

                mov   s,s wc ' sets C accordingly to the sign
                negc  r,r    ' negates result if necessary
```

The division requires a similar algorithm, but we subtract instead of add, **x = x / y**:

```
                mov   t,#16
                shl   y,#15
loop
                cmpsub x,y   wc
                rcl   x,#1
                djnz  t,#loop
```

The use of **cmpsub** reduces the amount of instructions per cycle loop to only 3, a nice bonus, if space is not a constraint these loops can be unrolled and up to 30% of time saved. If just 8 bits are used the first mov shoud be with 8 and the shift with 24.

Let's investigate **cmpsub** a bit more. As you know **cmp** is the **sub** instruction with the effect **nr** in place, do not write result back, to only affect flags. Flags as always must be explicitly indicated. **cmp** will rise **C** when the source is bigger than the destination. **cmpsub** will rise **C** if the source is *smaller* than the destination and will subtract the source to the destination placing the result into destination:

```
                cmpsub x,y   wc

x               long   5
y               long   7
```

Will **not** rise **C** neither will modify **x**.

```
                cmpsub x,y   wc
```

```
x               long   12
y               long   7
```

Will rise **C** and subtract **y** from **x**, resulting in a value of 5 in **x**.
This instruction can be exploited in some instances when a sequence like this is found:

```
                cmp    x,y   wc
        if_nc   sub    x,y   wc

x               long   5
y               long   7
```

It may not be a big difference, but a long saved here and there help when there is not that many of them.

The barrel shifter

The propeller has some neat tricks as we saw with **cmpsub**. It packs some more, and the barrel shifter is one of them. Most small processors, i.e. 8-bit ones and some 16 bit (z80, H8/300, HC11, ...) have simple 1-bit shifters. You may be familiar with the typical: let's convert a binary to a hex string requiring some 4 shifts per high digit. The cog in the propeller is not your average 8 bit processor, it is a 32 bit one, and a modern one!, so a barrel shifter was included, like in any serious (ARM, SH) processor. This shifter can perform a shift with any number of bits between 1 and 31 in exactly the same amount of time, because the shifter actually... has no shift registers!!, it has some multiplexors instead. Shown routines for multiply and divide make use of this, shifts of 8 or 16 bits in one instruction. I cannot stress enough how useful this is. The behaviour of the carry flag could seem a bit awkward, but it has its motives, (the zero flag is always set if the result is zero). The **C** flag is set if the *original* first or 31st bit was 1 depending on if it was a left shift (31st) or right shift (first), and also independently of it was a 10 bit shift or a 1 bit shift.

Simple binary multiplications by powers of two can be implemented using the shifter. Note that adding a value to itself has the same effect as a 1-bit left shift (**shl**), being the same instruction is some architectures.

The instruction **sar** shifts right arithmetically, that means it takes care of the sign. If a number is negative it will remain negative, if it is positive it will be positive till it reaches zero. Very useful to sign extend a number:

```
                shl     x,#16         ' shifts left, sign (bit 15) bec
omes bit 31
                sar     x,#16         ' shifts conserving the bit 31 s
```

```
tatus, sign extending the number to 32 bits
```

NOTE: In the case where two's complement numbers are used and they are not 32 bit quantities, i.e. the sign is other than bit 31, the method described above can be used to sign extend the number. A subsequent and safe call to **abs** will return the absolute value of the number. A use of **abs** before the number is sign extended will lead to an unmodified number.

The C and Z flags

The two flags available can be tested and modified in almost all instructions, provided that the corresponding effects are in place. The carry flag, **C**, indicates, depending on the instruction, a number of different things, parity, bit set or reset, carry, borrow, etc. In comparison instructions it indicates borrow (mostly). Sometimes it could be useful to set or reset it.

To set C we can do:

```
mov     x,#1
shr     x,#1    wc
```

To clear C we can do, this will not modify **x**, but will reset C (actually it mirrors the status of the 31st bit of x after the move):

```
mov     x,#0    wc nr
```

FFT

A working implementation, also written by me, of a Radix-2 FFT algorithm can be found here.

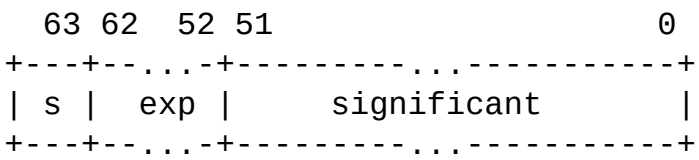
Fixed point math

An article about fixed point math can be found here. (it needs some more examples and descriptions, I'm working on that).

Double precision binary floating point

The single precision binary floating point library from the obex is known to almost everyone, but when single precision is not enough, double precision can solve the problem with its 53 bits of significant providing up to 16 decimal digits. Having a 32 bit ALU, the propeller can compute these numbers quite fast, using unrolled loops for multiplication and division, around 2000 cycles are required for either function.

The format as per the standard is



The significand sign is the bit 63, set when negative.
The exponent is biased with the number 1023, that means that all numbers greater than 0 have an exponent of 1023 or bigger. The exponent 2047 is used to represent Infinities and NaNs (Not a number). Infinities have a zeroed significand while NaNs have a non zero. An exponent of zero means either that the number is zero or that a denormalized number is presented. The support for denormalized numbers (those where their bit 53 is zero) is somewhat lacking in the following routines.
The significand is 53 bits long, but the most significant bit is assumed set when the exponent is non-zero and thus not stored.

Addition and subtraction are the simplest and fastest, adding 53 bits requires only 2 instructions, checking for bad input, scaling and sign management takes the rest. A possible implementation is as follows (LGPL v 2.0 code), see the link at the bottom for a file with all the routines.

```
' Adds two double precision numbers
' they should be already unpacked, result goes to R

dSUB            xor     rBSgn,cnt_h8     ' changes sign of B

dADD            test    flags,#FLG_NAN|FLG_INF   wz
       if_nz    call    #dLOADRNAN
       if_nz    jmp     #dADD_ret

                mov     rt1,rASgn
                xor     rt1,rBSgn   wz
       if_nz    jmp     #dSUB_1

                mov     rt1,rAExp
                subs    rt1,rBExp   wz
                abs     rt2,rt1
```

```
                mov     rRExp,rAExp
        if_z    jmp     #dADD_20

                cmp     rt1,#53   wc
        if_c    jmp     #dADD_5       ' shifts B
                cmp     rt2,#53   wc
        if_c    jmp     #dADD_10      ' shifts A
                cmp     rt1,#53   wc
        if_c    call    #dLOADBTOR
        if_nc   call    #dLOADATOR
                jmp     #dADD_ret

dADD_5          shr     rB,#1   wc
                rcr     rB1,#1
                djnz    rt2,#dADD_5
                jmp     #dADD_20

dADD_10         shr     rA,#1   wc
                rcr     rA1,#1
                djnz    rt2,#dADD_10
                mov     rRExp,rBExp

dADD_20         mov     rR1,rA1
                add     rR1,rB1   wc
                mov     rR,rA
                addx    rR,rB
                test    rR,cnt_bit54 wz
        if_nz   shr     rR,#1   wc
        if_nz   rcr     rR1,#1
        if_nz   add     rRExp,#1
                mov     rRSgn,rASgn

dADD_ret
dSUB_ret        ret

dSUB_1          mov     rt1,rAExp
                subs    rt1,rBExp   wz
                abs     rt2,rt1
         if_z   jmp     #dSUB_10    ' subs, no shift, exponents are equ
al
                cmp     rt1,#53   wc
         if_c   jmp     #dSUB_5    ' shifts B
                cmp     rt2,#53   wc
         if_c   jmp     #dSUB_5
                cmp     rt1,#53   wc
         if_c   call    #dLOADBTOR
```

```
        if_nc  call    #dLOADATOR
               jmp     #dSUB_ret

' exp of A is bigger than exp of B
dSUB_5         shr     rB,#1   wc
               rcr     rB1,#1
               djnz    rt2,#dSUB_5
' R=A-B
               mov     rRSgn,rASgn    ' transfers sign
               mov     rRExp,rAExp
               mov     rR1,rA1
               sub     rR1,rB1   wc, wz
               mov     rR,rA
               subx    rR,rB   wz
       if_nz   jmp     #dSUB_25
               jmp     #dSUB_35

' exponents are equal, so check significand
dSUB_10        cmp     rA1,rB1   wc, wz
               cmpx    rA,rB    wc, wz
    if_c       jmp     #dSUB_20          ' sig(A)<sig(B)
               jmp     #dSUB_35          ' numbers are equal

' B is bigger than A, we shift A and perform R=B-A
dSUB_15        shr     rA,#1   wc
               rcr     rA1,#1
               djnz    rt2,#dSUB_15
' R=B-A
dSUB_20        mov     rRSgn,rBSgn    ' transfers sign
               mov     rRExp,rBExp
               mov     rR1,rB1
               sub     rR1,rA1 wc
               mov     rR,rB
               subx    rR,rA

dSUB_25        mov     rt1,#53
dSUB_30        test    rR,cnt_bit53   wz
        if_nz  jmp     #dSUB_ret
               sub     rRExp,#1
               shl     rR1,#1   wc
               rcl     rR,#1
               djnz    rt1,#dSUB_30    ' normalizes
dSUB_35        call    #dLOADZTOR
               jmp     #dSUB_ret
```

Multplication requires a bit more work, a similar as that one explained before is used, but in two stages because a 96 bits partial result is kept instead of a full 107 when the most significant bits arer known to be zero. Exponents are added as usual and signs are xored.

```
' ******************************************************
' ****
' **** Multiplication R=A*B

dMUL            test    flags,#FLG_NAN|FLG_INF   wz
       if_nz    call    #dLOADRNAN
       if_nz    jmp     #dMUL_ret

                test    flags,#FLG_Z   wz
       if_z     call    #dLOADZTOR              ' if any of the numers a
re zero
       if_z     jmp     #dMUL_ret

                mov     rRExp,rAExp
                adds    rRExp,rBExp


' ** do not forget to check for overflow ;-)
                shl     rA,#11
                mov     rt1,rA1
                shl     rA1,#11
                shr     rt1,#21
                or      rA,rt1                  ' shifts to accomodate f
inal product

                mov     rt4,#32                 ' 32 bits first
                mov     rR,#0                   ' result significand
                mov     rR1,#0
                mov     rt1,#0
                mov     rt2,#0
                mov     rt3,#0
dMUL_10         shr     rB,#1   wc
                rcr     rB1,#1 wc
       if_nc    jmp     #dMUL_12
                add     rt2,rA1   wc
                addx    rt1,rA   wc
                addx    rR1,rt3   wc
dMUL_12         shl     rA1,#1   wc
                rcl     rA,#1  wc
                rcl     rt3,#1
```

```
               djnz    rt4,#dMUL_10
' 32 bits are done, now we multiply the other 21

               mov     rt2,#0
dMUL_15        shr     rB1,#1   wc
       if_nc   jmp     #dMUL_17
               add     rt1,rA   wc
               addx    rR1,rt3   wc
               addx    rR,rt2
dMUL_17        shl     rA,#1  wc
               rcl     rt3,#1 wc
               rcl     rt2,#1
               tjnz    rB1,#dMUL_15

               test    rR,cnt_bit53   wz
        if_nz  add     rRExp,#1               ' increments exponent
        if_z   shl     rt1,#1   wc
        if_z   rcl     rR1,#1   wc
        if_z   rcl     rR,#1

               shl     rt1,#1   wc            ' rounds up
               addx    rR1,#0   wc
               addx    rR,#0
               test    rR,cnt_bit54   wz
        if_nz  add     rRExp,#1               ' increments exponent
        if_nz  shr     rR,#1   wc
        if_nz  rcr     rR1,#1

dMUL_20        mov     rRSgn,rASgn
               xor     rRSgn,rBSgn
dMUL_ret       ret
```

Division needs also a loop, and a possible implementation follows:

```
' double division, R=A/B
'

dDIV           test    flags,#FLG_NAN|FLG_INF   wz
       if_nz   call    #dLOADRNAN
       if_nz   jmp     #dDIV_ret

               test    flags,#FLG_Z  wz
       if_z    jmp     #dDIV_2
               test    flags,#FLGB_Z   wz
```

```
       if_nz   or      flags,#FLG_ERR_DIV0    ' x/0 (even 0/0)
       if_z    call    #dLOADZTOR             ' 0/x = 0
               jmp     #dDIV_ret

dDIV_2         mov     rRExp,rAExp
               subs    rRExp,rBExp

               shl     rA,#11
               mov     rt1,rA1
               shl     rA1,#11
               shr     rt1,#21
               or      rA,rt1                 ' shifts to accomodate f
inal product

               shl     rB,#11
               mov     rt1,rB1
               shl     rB1,#11
               shr     rt1,#21
               or      rB,rt1                 ' shifts to accomodate f
inal product

               mov     rt1,#53
               cmp     rA1,rB1   wc, wz
               cmpx    rA,rB   wc, wz
       if_c    jmp     #dDIV_4

       if_nz   sub     rt1,#1
               sub     rA1,rB1   wc
               subx    rA,rB

dDIV_4         shr     rB,#1   wc
               rcr     rB1,#1
               sub     rRExp,#1

               mov     rR1,#0
               mov     rR,#0   wc
dDIV_5         cmp     rA1,rB1   wc,wz
               cmpx    rA,rB   wc
       if_c    jmp     #dDIV_10
               sub     rA1,rB1   wc
               subx    rA,rB   wc

dDIV_10        rcl     rR1,#1   wc
               rcl     rR,#1   wc
               shl     rA1,#1  wc
               rcl     rA,#1
```

```
              djnz    rt1,#dDIV_5

              xor     rR,cnt_f
              xor     rR1,cnt_f

dDIV_20       and     rR,cnt_sigh ' clears garbled bits

dDIV_ret      re
```

A comparsion routine could be implemented as follows:

```
' Comapres A and B, sets the flags accordingly.
' Two NaNs will give a non equal result

dCMP          test    flags,#FLG_NAN|FLG_INF   wz
      if_nz   mov     rt1,#2   wz, wc
      if_nz   jmp     #dCMP_ret   ' two NaNs or Infs give a diff resu
lt

              cmp     rBSgn,rASgn wz, wc
      if_nz   jmp     #dCMP_ret   ' different signs, neg < pos

              cmps    rAExp,rBExp   wz, wc
      if_nz   jmp     #dCMP_ret   ' they are different
              cmp     rA1,rB1   wz, wc
              cmpx    rA,rB   wz, wc
dCMP_ret      ret
```

Some support code is also neede, to pack and unpack numbers, to set flags and so on.

```
' loads A from pointer ptr and unpacks

dLOADA        andn    flags,#FLGA_Z|FLGA_INF|FLGA_NAN
              rdlong  rA,ptr
              add     ptr,#4
              mov     rAExp,rA
              rdlong  rA1,ptr
              mov     rASgn,rA
              and     rASgn,cnt_h8
              add     ptr,#4

              andn    rA,cnt_sexp   wz
```

```
       if_z    mov     rA1,rA1   wz
       if_z    or      flags,#FLGA_Z       ' temporal Z flag if signif
icand is zero

               and     rAExp,cnt_exp
               cmp     rAExp,cnt_exp   wc ' checks for NaN or Inf
       if_nc   test    flags,#FLGA_Z   wz
   if_nc_and_z or      flags,#FLGA_INF     ' infinity
  if_nc_and_nz or      flags,#FLGA_NAN     ' Not a number

               shr     rAExp,#20   wz
       if_nz   or      rA,cnt_bit53       ' adds implicit 53 bit if no
t zero
       if_nz   sub     rAExp,cnt_bias     ' subtract bias if is not ze
ro
       if_nz   andn    flags,#FLGA_Z       ' is not zero anymore
dLOADA_ret     ret

' loads B from pointer ptr and unpacks

dLOADB         andn    flags,#FLGB_Z|FLGB_INF|FLGB_NAN
               rdlong  rB,ptr
               add     ptr,#4
               mov     rBExp,rB
               rdlong  rB1,ptr
               mov     rBSgn,rB
               and     rBSgn,cnt_h8
               add     ptr,#4

               andn    rB,cnt_sexp   wz
       if_z    mov     rB1,rB1   wz
       if_z    or      flags,#FLGB_Z       ' temporal Z flag if signif
icand is zero

               and     rBExp,cnt_exp
               cmp     rBExp,cnt_exp   wc ' checks for NaN or Inf
       if_nc   test    flags,#FLGB_Z   wz
   if_nc_and_z or      flags,#FLGB_INF     ' infinity
  if_nc_and_nz or      flags,#FLGB_NAN     ' Not a number

               shr     rBExp,#20   wz
       if_nz   or      rB,cnt_bit53       ' adds implicit 53 bit if no
t zero
       if_nz   sub     rBExp,cnt_bias     ' subtract bias if is not ze
ro
       if_nz   andn    flags,#FLGB_Z       ' is not zero anymore
```

```
dLOADB_ret     ret

' packs and saves R to ptr, destroys rR, rRExp

dSAVER         andn    rR,cnt_bit53     ' removes imlicit 1 at bit 53
               or      rR,rRSgn
               test    rRExp,rRExp   wz
       if_nz   add     rRExp,cnt_bias   ' if it is zero, keep it zero
               shl     rRExp,#20
               or      rR,rRExp
               wrlong  rR,ptr
               add     ptr,#4
               wrlong  rR1,ptr
               add     ptr,#4
dSAVER_ret     ret

' loads NaN into R
' exp = 0x3ff
' non-zero significand

dLOADRNAN      mov     rRExp,cnt_NaN
               mov     rR,cnt_bit52
               mov     rR1,#0
dLOADRNAN_ret  ret
' loads A into R
dLOADATOR      mov     rR,rA
               mov     rR1,rA1
               mov     rRExp,rAExp
               mov     rRSgn,rASgn
dLOADATOR_ret  ret
' loads B into R
dLOADBTOR      mov     rR,rB
               mov     rR1,rB1
               mov     rRExp,rBExp
               mov     rRSgn,rBSgn
dLOADBTOR_ret  ret

' loads zero into R
dLOADZTOR      mov     rR,#0
               mov     rR1,#0
               mov     rRExp,#0
               mov     rRSgn,#0
dLOADZTOR_ret  ret


' Numbers are unpacked for easier manipulation
```

```
' explicit 53ed bit is added


rA              long    $10_0000
rA1             long    0'$5555_5555
rAExp           long    $3ff
rASgn           long    0

rB              long    $10_0000
rB1             long    0'$5555_5555
rBExp           long    $3ff
rBSgn           long    0

rR              long    0
rR1             long    0
rRExp           long    0
rRSgn           long    0

rt1             long    0
rt2             long    0
rt3             long    0
rt4             long    0

rt5             long    0

ptr             long    0
flags           long    0

cnt_h8          long    $8000_0000
cnt_exp         long    $7ff0_0000
cnt_sexp        long    $fff0_0000
cnt_bit54       long    $0020_0000
cnt_bit53       long    $0010_0000
cnt_bit52       long    $0008_0000
cnt_bias        long    $3ff
cnt_NaN         long    $7ff
cnt_f           long    $ffff_ffff
cnt_sigh        long    $1f_ffff
```

BCD MATH

As everyone knows BCD stands for binary coded decimal, i.e. each decimal digit is represented in binary, and all operations are done with decimal numbers... that means that from our stand point: exactly as we will do them on paper. BCD arithmetic has some caveats and some advantages compared to binary arithmetic:

Pros:

- No rounding problems when working with human-readable numbers
- Fastest floating-point to string conversion
- I like it
- BCD floating point

Cons

- Slower
- wastes space
- poor support in the assembler

Number representation

To represent every decimal digit, a minimum of 4 bits are needed. That means: for 8 digits a long (32 bits) will be needed, while with a binary representation a long would hold 9 or 10 digits.

```
For a 8 digit number 12345678

+---+---+---+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
+---+---+---+---+---+---+---+---+

For the same number the binary would have been BC614E (24 bits instead
 of 32)
```

The four basic arithmetic operations are performed on BCD numbers as we will do them on paper, but the implementations are not that straightforward because carry/borrow from digit to digit has to be considered, something we hardly think about with binary numbers, i.e. the addition produces a number greater than 9. Binary operations have to be used because the propeller lacks BCD ones and **daa** (decimal adjust after addition as found in the Z80, HC11, etc) or similar. A compare and add/subtract if condition method has to be used operating on a digit-by-digit basis.

```
' Adds two 8 digit numbers (longs)
' carry is used in negative logic !
```

```
mADD8           mov     rcarry,#3
                shr     rcarry,#1 wc ' sets carry flag
mADD8C          mov     rmsk1,#$f
                mov     rt5,#0
                mov     rsh1,#10

mADD8_1         mov     rt3,rt1
                and     rt3,rmsk1

                mov     rt4,rt2
                and     rt4,rmsk1

        if_nc   add     rt4,rcarry
                add     rt3,rt4 wc
        if_nc   jmp     #mADD8_5
                mov     rcarry,#1 wc ' clears carry flag for next roun
d
                jmp     #mADD8_ret

mADD8_5         cmp     rt3,rsh1 wc
        if_nc   sub     rt3,rsh1
                or      rt5,rt3
                rol     rcarry,#4 ' magic
                shl     rsh1,#4
                shl     rmsk1,#4 wz
        if_nz   jmp     #mADD8_1
mADD8C_ret
mADD8_ret       ret
```

The code operates over longs, i.e. 8 digits, **rt1** is then added to **rt2** with result on **rt5**. Several helping variables are then needed: **rmsk1** is the digit mask that after every cycle is shifted left and also used as counter, **rcarry** is the number that will be added to the digit that is left of the current added, **rt3** is the current digit from **rt1** and **rt4** is from **rt2**, **rsh1** is the overflow value to compare with that is also shifted left after every cycle.

As you can see this small routine replaces the comfortable and *short* binary **add** :).

The subtraction could be implemented like this:

```
' Subs two 8 digit numbers (longs)

mSUB8           mov     rcarry,#1 wc ' clears carry flag
mSUB8C          mov     rmsk1,#$f
                mov     rt5,#0
```

```
                mov     rsh1,#10
mSUB8_1         mov     rt3,rt1
                and     rt3,rmsk1
                mov     rt4,rt2
                and     rt4,rmsk1
        if_c    add     rt4,rcarry
                sub     rt3,rt4 wc

        if_c    add     rt3,rsh1
                or      rt5,rt3
                rol     rcarry,#4 ' magic
                shl     rsh1,#4
                shl     rmsk1,#4 wz
        if_nz   jmp     #mSUB8_1
mSUB8C_ret
mSUB8_ret       ret
```

This routine uses the same tricks and variables as before but the **C** flag is used in positive logic instead.

Both routines can be concatenated calling mADD8/mSUB8 first and mADD8/mSUB8C later without modifying the **C** flag in between.

To multiply the method of shift-add as we will do by hand is one of the few resources left. Sadly, in comparison with binary multiplication, the *short* **add** has to be replaced with a call to mADD8, but the logic is similar. Here **rA:rA1** is multiplied by **rB1** using the pair **rR**:**rR1** as result.

```
' multiplies A*rB1
'
' uses rt1, rt2, rt5, rt6, rcnt1, rp, rB1, rR, rR1
'
' clogged by mADD8 rt3, rt4, rt5, rcarry, rmsk1, rsh1, rR, rR1

mMUL8           mov     rt7,#8

mMUL8_5         mov     rcnt1,rB1
                and     rcnt1,#$f wz
                mov     rt6,#0
        if_z    jmp     #mMUL8_15

mMUL8_10        mov     rt1,rA1
                mov     rt2,rR1
                call    #mADD8
                mov     rR1,rt5
                mov     rt1,rA
```

```
                mov     rt2,rR
                call    #mADD8C
                mov     rR,rt5
        if_nc   add     rt6,#1          ' carry counter
                djnz    rcnt1,#mMUL8_10

mMUL8_15        mov     rt5,rR
                shl     rt5,#4
                shr     rR1,#4
                or      rR1,rt5         ' shift right rR:rR1
                ror     rt6,#4          ' convert to MSD
                or      rR,rt6          ' sets new carry digit
                shr     rB1,#4
                djnz    rt7,#mMUL8_5
mMUL8_ret       ret
```

What the small fragment of code does is to add **rA:rA1** to itself as many times as the digits in **rB1** from right to left say. **mADD8** and **mADD8C** take care of adding **rA:rA1** to **rR:rR1**. after each digit of **rB1**, **rR:rR1** is shifted right to discard the rightmost digit and to make place for the new MSD.

All this may seem like a real waste, but opens the door to BCD-floating point, the real end.

Floating point BCD (BCD12)

To complete the package, we should talk about how to operate on whole floating point numbers. For that we will consider the following notation (which we will call BCD12 from now on), (more possibilities are of course available, and the principles explained here apply):

```
         +---+---+---+---+---+---+---+---+
long 0 : | S |MSD| A | 9 | 8 | 7 | 6 | 5 |
         +---+---+---+---+---+---+---+---+
         +---+---+---+---+---+---+---+---+
long 1 : | 4 | 3 | 2 | 1 |LSD| E2| E1| E0|
         +---+---+---+---+---+---+---+---+
```

The floating point number occupies 2 longs in HUB memory (4 in COG memory) and is packed according to the diagram above, each cell represents a nibble (4 bits).
S is the significant sign, MSD to LSD are the significant digits, 12 in total, and the exponent occupies the last three nibbles. The exponent is a two's complement 12 bit number. Negative numbers represent negative powers of 10.

The number Zero is represented as two zeroed longs.

To better exploit the propeller capabilities, the HUB representation is unpacked to 4 longs in COG memory (it may seem a waste, but the access to the different parts in every routine saves many more longs than the 2 extra used for the unpacked representation):

```
Representation in cog's memory.
         +---+---+---+---+---+---+---+---+
long 0 : | 0 |MSD| A | 9 | 8 | 7 | 6 | 5 |  rA, rB, rR
         +---+---+---+---+---+---+---+---+
         +---+---+---+---+---+---+---+---+
long 1 : | 4 | 3 | 2 | 1 |LSD| 0 | 0 | 0 |  rA1, rB1, rR1
         +---+---+---+---+---+---+---+---+
         +---+---+---+---+---+---+---+---+
long 2 : | es| es| es| es| es| E2| E1| E0|  rAExp, rBExp, rRExp
         +---+---+---+---+---+---+---+---+
         +---+---+---+---+---+---+---+---+
long 3 : | S | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  rASgn, rBSgn, rRSgn
         +---+---+---+---+---+---+---+---+
```

As you can see this representation keeps the form of the packed version, for easier access. The exponent is sign extended with the **shl/sar** combination as seen in the load routines below. This allows for fast add/compare/subtract of exponents.

A conversion routine that loads **rA:rA1**, **rAExp** and **rASgn** could be this one, using **ptr1** as source pointer:

```
' Loads A from a BCD12
LOADA           rdlong rA,ptr1        ' reads first long
                add    ptr1,#4
                mov    rASgn,rA
                rdlong rA1,ptr1       ' reads 2nd long
                and    rASgn,cnt_SMASK
                andn   rA,cnt_SMASK
                mov    rAExp,rA1
                shl    rAExp,#20    ' exponent is signed
                sar    rAExp,#20
                and    rA1,cnt_2LMASK
LOADA_ret       ret

cnt_SMASK       long    $8000_0000    ' sign mask
```

```
cnt_D12         long    $0f00_0000    ' MSD (digit 12) mask
cnt_2LMASK      long    $ffff_f000    ' low long mask
```

The packing of a result to HUB memory could be implemented as below using **ptr1** as destination pointer:

```
' saves R as a BCD12
SAVER           mov    rt1,rR
                or     rt1,rRSgn
                mov    rt2,rRExp
                andn   rt2,cnt_2LMASK
                mov    rt3,rR1
                and    rt3,cnt_2LMASK
                wrlong rt1,ptr1
                or     rt2,rt3
                add    ptr1,#4
                wrlong rt2,ptr1
SAVER_ret       ret
```

The packing scheme now seems to fit nicely. The extra empty digit at the left of the MSD is used as guard digit during addition and multiplication, and thus plays to our advantage. The empty digits at the right of the LSD (cleared using the cnt_2LMASK) pose a performance penalty due to its computation in **mADD8 /mSUB8**, the most significant of the group could be used for instance, to better round results.

From ASCII to BCD12

To convert a string to a BCD12 (or for that matter to binary floating point) a set of rules, like always, should be put into action. These rules help to determine what can be accepted as a valid input and what is not.

- Spaces preceding the first valid character should be ignored. No spaces are allowed in between.
- The only valid characters are digits 0 to **9**, signs **+** and **-**, the period **.** and the letter **e**.
- An optional significant sign, it should be the first valid character if present.
- The next valid symbol is either a digit or the period.
- A number of digits (any digit present beyond 12 will be chopped but they may add to the exponent if the number has an exponent greater than 12), if only a period was present a minimum of one digit must be present.
- An exponent composed of three parts, the letter **e** to indicate it, an optional sign **+** or **-** and a minimum of 1 digit to a maximum of three digits.

All these numbers must be valid then:

-0.0012
123400000
0000123000
+45e-123
-0.0000004e+40 (why someone will write a number like this is beyond me)

Some invalid combination include:

-e+4 (No significant digit(s))
a1e-4 (invalid char)
4.5e- (No exponent digit(s))
. (No significant digit(s))

Will those rules we should be able to write some nice code. Note: The use of assembler for this purpose is... not recommendable as this routine will be not only complicated but over all, long, and not time critical at all. But as an exercise is a good one.

```
ASCIITOBCD

ASCIITOBCD_ret  ret
```

From BCD12 to ASCII

Conversion from BCD12 to ASCII is quite straightforward, but some points should be noted. To how many places do we have to represent the number ?, that is the question to be asked. Complementary of that are we going to use all the available range or just a small subset of numbers ?. As the propeller has a limited COG memory, a small and specifically tailored conversion may be the way to go.

Small and tailored

A small and tailored conversion can be seen as a quick-and-dirty approach, let's say that the numbers to represent are in the tens of thousands, decimals are unimportant, then we can truncate them without looking back. A possibility could be:

```
' Converts a number in the 10000 to 99999 range to ascii
' Number is in r, sign and decimals are unimportant
' ptr1 is destination
'
```

```
TOASCIIQD       mov    rRExp,rRExp wc 'checks for negative exponent,
       if_c     mov    rt1,#48
       if_c     call   #EMITASCII
       if_c     jmp    #TOASCIIQD_40
' number may be in range
                cmp    rRExp,#5 wc
       if_nc    mov    rt1,#69 ' E signals error
       if_nc    call   #EMITASCII
       if_nc    jmp    #TOASCIIQD_40
 ' number is in range
                mov    rt2,rRExp
                add    rt2,#1
                mov    rt3,rR ' working significant
TOASCIIQD_10    mov    rt1,rt3
                shr    rt1,#24
                and    rt1,#15
                add    rt1,#48 ' converts digit to ASCII
                call   #EMITASCII
                shl    rt3,#4
                djnz   rt2,#TOASCIIQD_10

TOASCIIQD_40    mov    rt1,#0
                call   #EMITASCII
TOASCIIQD_ret   ret

' writes an ascii to HUB and increments pointer
' rt1 is the byte to write
' ptr1 is the pointer

EMITASCII       wrbyte rt1,ptr1
                add    ptr1,#1
EMITASCII_ret   ret
```

The helper routine **EMITASCII** does... well exactly that!, and increments the pointer. Small helping routines can save a few longs here and there, in some situations, but they can increase execution by 8 cycles each time they are called. So for simple and short loops it is the way to go.

If we were to consider rounding things may get... slower, and longer. The previous example can be taken as rounded with the floor function, or simply put: truncation. Rounding to nearest can be implemented adding 0.5. A simple call to **mADD8** with the properly formatted argument can be used... something like this:

```
 ' Converts a number in the 10000 to 99999 range to ascii
 ' Number is in r, sign is unimportant. Rounding is performed if the n
```

```
umber is >= 1
 ' ptr1 is destination
 '
TOASCIIQDR      mov    rRExp,rRExp wc 'checks for negative exponent,
        if_c    mov    rt1,#48
        if_c    call   #EMITASCII
        if_c    jmp    #TOASCIIQDR_40
 ' number may be in range
                cmp    rRExp,#5 wc
TOASCIIQDR_5
        if_nc   mov    rt1,#69 ' E signals error
        if_nc   call   #EMITASCII
        if_nc   jmp    #TOASCIIQDR_40
' number is in range, adds rounding factor
                mov    rt1,rR
                mov    rt2,#5 ' rounding argument
                mov    rt3,#5
                sub    rt3,rRExp
                shl    rt3,#2
                shl    rt2,rt3 ' adjust rounding digit
                call   #mADD8
                mov    rt2,rRExp
                test   rt5,cnt_MSD wz
        if_nz   add    rt2,#1 ' increments exponent if overflow
        if_nz   shr    rt5,#4 ' shifts working significant one to the
right
                cmp    rt2,#5 wc
        if_nc   jmp    #TOASCIIQDR_5
                add    rt2,#1
TOASCIIQDR_10   mov    rt1,rt5
                shr    rt1,#24
                and    rt1,#15
                add    rt1,#48 ' converts digit to ASCII
                call   #EMITASCII
                shl    rt5,#4
                djnz   rt2,#TOASCIIQDR_10

TOASCIIQDR_40   mov    rt1,#0
                call   #EMITASCII
TOASCIIQDR_ret  ret

 ' writes an ascii to HUB and increments pointer
 ' rt1 is the byte to write
 ' ptr1 is the pointer

EMITASCII       wrbyte rt1,ptr1
```

```
                add    ptr1,#1
EMITASCII_ret   ret
```

If all numbers have to be converted, i.e. the full range at full precision, 12 digits, plus sign and exponent a maximum of 19 characters will be generated. Shorter representations, when applicable, are possible depending on the number. A good criteria is if the number of digits to represent is greater than say 12, a number with exponent should be used. It is easier to read 1234 than 1.234e+3. The last notation will be called scientific notation. A smart routine that can differentiate between this two cases has three parts. The first is the conversion to scientific if the exponent is greater than 12 in any direction.

Note: The BCD12 representation of numbers in following examples (the ones on the left) are written in scientific notation for clarity.

The second is the representation of numbers smaller than one, where zeroes should be inserted between the decimal point and the first digit of the significant, the number of zeroes in this case is the absolute value of the exponent minus 1.

The third and final case is that of numbers greater than 1 that have 12 or less digits, zeroes should be removed after the decimal point if there are no significant digit at the right, and zeroes should be inserted between the last digit at the right and the decimal point if required (this last case is the more complex one of the three).

Number	Representation	Comment
1.34e100	1.34e100	No prettier form available, exponent > 12
3.4e-1	.34	
3.4e-5	.000034	In this case some zeroes where added after the decimal point. Those zeroes are not in the original BCD12 number, because they are all normalized
1.2e1	12	not like 1.2e1 or 12.0000000000
1.234e1	12.34	
4.e+6	4000000	In this case zeroes where added after the last significant digit, the 4, and the decimal point.

Now, let's see some code:

```
TOASCII

TOASCII_ret     ret
```

Addition and subtraction of BCD12 numbers

The routine shown before allows for addition of two 8 BCD numbers. Based on this, and with some improvements for speed and size a successful implementation of a complete BCD12 plus BCD12 addition/subtraction would be something like this:

```
kkBCDSUB        xor     rBSgn,cnt_SMASK    ' I love long routines ;-)
kkBCDADD        mov     rt1,rASgn
                xor     rt1,rBSgn
                test    rt1,cnt_SMASK wz
        if_nz   jmp     #kkSUB
' falls to ADD15
' ****************************************
' ***
' *** addition of two bcd unpacked numbers rR = rA + rB

kkADD           mov     rt1,rAExp
                subs    rt1,rBExp wz
                abs     rt2,rt1
        if_z    jmp     #kkADD_20        ' adds, no shift
                cmp     rt1,#16 wc
        if_c    jmp     #kkADD_5        ' shifts B
                cmp     rt2,#16 wc
        if_c    jmp     #kkADD_10

                cmp     rt1,#16 wc
        if_c    call    #LOADBTOR
        if_nc   call    #LOADATOR
                jmp     #kkADD_ret

kkADD_5         call    #kkmSHRB15
                djnz    rt2,#kkADD_5
                mov     rRExp,rAExp          ' exponent of A
                jmp     #kkADD_20

kkADD_10        call    #kkmSHRA15
                djnz    rt2,#kkADD_10
```

```
                mov     rRExp,rBExp     ' exponent of B
kkADD_20        movs    kkmADD8_1,#rA1
                movs    kkmADD8_2,#rB1
                call    #kkmADDSM
                mov     rR1,rt6
                mov     rR,rt5
                and     rt5,cnt_MSD  wz
        if_nz   call    #kkmSHRR15
        if_nz   add     rRExp,#1
                mov     rRSgn,rASgn     ' sets sign from A
                jmp     #kkADD_ret

' ******************************************
' ***
' *** Substraction
' ***


kkSUB           mov     rt1,rAExp
                subs    rt1,rBExp wz
                movs    kkmSUB8_1,#rB1
                movs    kkmSUB8_2,#rA1          ' prepares for R=B-A
                abs     rt2,rt1
        if_z    jmp     #kkSUB_15       ' adds, no shift
                cmp     rt1,#16 wc
        if_c    jmp     #kkSUB_10       ' shifts B
                cmp     rt2,#16 wc
        if_c    jmp     #kkSUB_5
                cmp     rt1,#16 wc
        if_c    call    #LOADBTOR
        if_nc   call    #LOADATOR
                jmp     #kkSUB_ret

' B is bigger than A, we shift A and perform R=B-A
kkSUB_5         call    #kkmSHRA15
                djnz    rt2,#kkSUB_5
                jmp     #kkSUB_20

' exponents are equal, so check significand
kkSUB_15        call    #kkmCMP15
        if_c    jmp     #kkSUB_20           ' sig(A)<sig(B)
                jmp     #kkSUB_17
' A is bigger than B
kkSUB_10        call    #kkmSHRB15
                djnz    rt2,#kkSUB_10
kkSUB_17        movs    kkmSUB8_1,#rA1
```

```
                movs    kkmSUB8_2,#rB1
kkSUB_20
                mov     rRSgn,rASgn    ' transfers sign
                mov     rRExp,rAExp
                call    #kkmSUBSM
                mov     rR1,rt6
                mov     rR,rt5

kkSUB_25        and     rt5,cnt_D12 wz
        if_nz   jmp     #kkSUB_ret
                sub     rRExp,#1
                call    #kkmSHLR15
                call    #kkmCMPRZ    ' tests for zero
        if_nz   jmp     #kkSUB_25
                call    #LOADZTOR
kkSUB_30
kkADD_ret
kkBCDSUB_ret
kkBCDADD_ret
kkSUB_ret       ret

' Adds two numbers
kkmADDSM        call    #kkmADD8
                mov     rt6,rt5
                sub     kkmADD8_1,#1
                sub     kkmADD8_2,#1
                call    #kkmADD8C
kkmADDSM_ret    ret

kkmSUBSM        call    #kkmSUB8
                mov     rt6,rt5
                sub     kkmSUB8_1,#1
                sub     kkmSUB8_2,#1
                call    #kkmSUB8C
kkmSUBSM_ret    ret

' Adds two 8 digit longs
' carry is used in negative logic !

kkmADD8         mov     rcarry,#1   wc ' clears carry
                mov     rmsk1,#$f
kkmADD8C        mov     rt5,#0
                mov     rsh1,#10

kkmADD8_1       mov     rt3,0-0        '
                and     rt3,rmsk1
```

```
kkmADD8_2       mov     rt4,0-0
                and     rt4,rmsk1

         if_c   add     rt4,rcarry
                add     rt3,rt4   wc
         if_c   add     rt3,cnt_SIX          ' adds to convert to deci
mal if rightmost digit

kkmADD8_5 if_nc cmpsub  rt3,rsh1   wc
                or      rt5,rt3
                rol     rcarry,#4 ' magic
                rol     rmsk1,#4
                shl     rsh1,#4   wz
         if_nz  jmp     #kkmADD8_1
kkmADD8C_ret
kkmADD8_ret     ret

' Subs two 8 digit longs

kkmSUB8         mov     rcarry,#1   wc ' clrs carry flag
                mov     rmsk1,#$f
kkmSUB8C        mov     rt5,#0
                mov     rsh1,#10
kkmSUB8_1       mov     rt3,0-0
                and     rt3,rmsk1
kkmSUB8_2       mov     rt4,0-0
                and     rt4,rmsk1
       if_c     add     rt4,rcarry
                sub     rt3,rt4   wc

       if_c     add     rt3,rsh1
                or      rt5,rt3
                rol     rcarry,#4 ' magic
                rol     rmsk1,#4
                shl     rsh1,#4   wz
       if_nz    jmp     #kkmSUB8_1
kkmSUB8C_ret
kkmSUB8_ret     ret

' loads A to R
LOADATOR        mov    rR,rA
        mov    rR1,rA1
        mov    rRExp,rAExp
        mov    rRSgn,rBSgn
LOADATOR_ret    ret
```

```
' loads B to R
LOADBTOR        mov    rR,rB
        mov    rR1,rB1
        mov    rRExp,rBExp
        mov    rRSgn,rBSgn
LOADBTOR_ret    ret
' loads zero to R
LOADZTOR        mov    rR,#0
        mov    rR1,#0
        mov    rRExp,#0
        mov    rRSgn,#0
LOADZTOR_ret    ret
```

The subtraction is implemented wrapping the addition with a sign change for the subtraend. Note that there are actually three different stages. the first one represented by **kkBCDADD** (and **kkBCDSUB**) are the routines you should call when **rA** and **rB** (and the other related variables) have been loaded with the **LOADA** and **LOADB** routines described avobe.
kkADD is called when the numbers (BCD12) should be added, i.e. when the sign of both are equal. To add the two parts then **kkmADDSM** is called. This last routine operates over **rA**:**rA1** and **rB**:**rB1** as if they where 16 digit numbers (ignoring the fact that some digits are always zero).

This code uses self-modifying techniques. This saves some longs used by variables and also some time. Compared to the previous routine **kkmADD8** is 4 longs shorter and uses less variables, and the nice **cmpsub** instruction. The carry is now used in positive logic, i.e. a carry set means that the last add gave carry, (contrary to previous use).

Multiplication

Using a modified **MUL8** routine, to support the self-modifying version of kkmADD8, a full multiplication ca be easily implemented as shown below.

```
' ******************************************************
' ****
' **** Multiplication R=A*B

kkBCDMUL        mov     rRExp,rAExp
                adds    rRExp,rBExp
' ** do not forget to check for overflow ;-)

                mov     rR,#0                   ' result significand
```

```
                mov     rR1,#0
                test    rB1,rB1 wz
        if_nz   call    #kkmMUL8                ' avoid 8 zeroes if nec
essary
kkBCDMUL_5      mov     rB1,rB
                call    #kkmMUL8

                call    #kkmSHLR15
                mov     rt1,rR
                and     rt1,cnt_D12 wz
        if_nz   add     rRExp,#1                ' increments exponent
        if_z    call    #kkmSHLR15              ' normalizes significa
nd

                mov     rRSgn,rASgn
                xor     rRSgn,rBSgn
kkBCDMUL15_ret    ret

'
' multiplies A*rB1
'
' uses rt1, rt2, rt5, rt6, rcnt1, rp, rB1, rR, rR1
'
' clogged by mADD8 rt3, rt4, rt5, rcarry, rmsk1, rsh1
' clogged by mSHRR15 rt5, rR, rR1

kkmMUL8         mov     rt7,#8

kkmMUL8_5       mov     rcnt1,rB1
                and     rcnt1,#$f wz
                mov     rt6,#0
        if_z    jmp     #kkmMUL8_15

kkmMUL8_10      movs    kkmADD8_1,#rA1
                movs    kkmADD8_2,#rR1
                call    #kkmADD8
                mov     rR1,rt5
                movs    kkmADD8_1,#rA
                movs    kkmADD8_2,#rR
                call    #kkmADD8C
                mov     rR,rt5
        if_c    add     rt6,#1          ' carry counter
                djnz    rcnt1,#kkmMUL8_10

kkmMUL8_15      mov     rt5,#4
kkmMUL8_20      shr     rR,#1 wc
```

```
                rcr     rR1,#1
                djnz    rt5,#kkmMUL8_20
                ror     rt6,#4          ' convert to MSD
                or      rR,rt6          ' sets new carry digit
                shr     rB1,#4
                djnz    rt7,#kkmMUL8_5
kkmMUL8_ret     ret
```

Division

To divide we implement a similar algorithm as before, i.e. the same algorithm you learned at school. Some helping routines are necessary, shifts and compares. Let's see the code, but before we should note that x/0 and 0/0 will give some errors, in **rerr**.

```
.section COG cog0 ' needed for pPropellerSim (like DAT/org combination
)
'
'
'
' BCD12 DIVISION
'
' A/B
' Destroys rA:rA1 rB:rB1

ERR_DIV0 = 1
ERR_DIV00 = 2


' ******************************************************
' ****
' ****
' **** Division R = A / B
' ****


kkBCDDIV        call    #kkmCMPBZ
        if_z    mov     rerr,#ERR_DIV0
        if_z    jmp     #kkBCDDIV_ret
                call    #kkmCMPAZ
        if_z    mov     rerr,#ERR_DIV00
        if_z    jmp     #kkBCDDIV_ret
' real division, exponent and sign
                mov     rR,#0
                mov     rR1,#0
```

```
                mov     rRExp,rAExp
                subs    rRExp,rBExp
                mov     rRSgn,rASgn
                xor     rASgn,rBSgn
                mov     rt7,#12     ' number of digits
                call    #kkmCMP15         ' compares A with B
        if_c    call    #kkmSHLA15
        if_c    subs    rRExp,#1         ' decrements exponent if A<B

kkBCDDIV_20     call    #kkmCMP15
        if_c    jmp     #kkBCDDIV_30
                movs    kkmSUB8_1,#rA1
                movs    kkmSUB8_2,#rB1
                call    #kkmSUB8
                mov     rA1,rt5
                movs    kkmSUB8_1,#rA
                movs    kkmSUB8_2,#rB
                call    #kkmSUB8C
                mov     rA,rt5
        if_nc   add     rR1,#1         ' increments count
                jmp     #kkBCDDIV_20

kkBCDDIV_30     call    #kkmSHLA15         ' if borrow, shift left
                call    #kkmSHLR15         ' shift for next digit
                djnz    rt7,#kkBCDDIV_20
                call    #kkmSHLR15
                call    #kkmSHLR15         ' adjusts result
kkBCDDIV_ret    ret

' shifts A left one digit
kkmSHLA15       mov     rt5,rA1
                shl     rA1,#4
                shl     rA,#4
                shr     rt5,#28
                or      rA,rt5
kkmSHLA15_ret   ret

' shifts A right one digit
kkmSHRA15       mov     rt5,rA
                shr     rA,#4
                shl     rt5,#28
                shr     rA1,#4
                or      rA1,rt5
kkmSHRA15_ret   ret

' shifts B left one digit
```

```
kkmSHLB15       mov      rt5,rB1
                shl      rB1,#4
                shl      rB,#4
                shr      rt5,#28
                or       rB,rt5
kkmSHLB15_ret   ret

' shifts B right one digit
kkmSHRB15       mov    rt5,rB
                shr    rB,#4
                shl    rt5,#28
                shr    rB1,#4
                or    rB1,rt5
kkmSHRB15_ret   ret

' shifts B left one digit
kkmSHLR15       mov      rt5,rR1
                shl      rR1,#4
                shl      rR,#4
                shr      rt5,#28
                or       rR,rt5
kkmSHLR15_ret   ret

' shifts R right one digit
kkmSHRR15       mov     rt5,rR
                shr     rR,#4
                shl     rt5,#28
                shr     rR1,#4
                or      rR1,rt5
kkmSHRR15_ret   ret

kkmCMP15        cmp     rA,rB   wc wz
        if_z    cmp     rA1,rB1   wc wz
kkmCMP15_ret    ret

kkmCMPRZ        test    rR,rR   wz
        if_z    test    rR1,rR1   wz
kkmCMPRZ_ret    ret

' checks if A or B are zero
kkmCMPAZ        test    rA,rA   wz
        if_z    test    rA1,rA1   wz
kkmCMPAZ_ret    ret

kkmCMPBZ        test    rB,rB   wz
        if_z    test    rB1,rB1   wz
```

```
kkmCMPBZ_ret    ret

cnt_SMASK       long    $8000_0000    ' sign mask
cnt_D12         long    $0f00_0000    ' MSD (digit 12) mask
cnt_2LMASK      long    $ffff_f000    ' low long mask
```

Square root

The calculus of the square root can be performed using a plurality of methods, while just some of them are useful for computers others are useful for calculation by hand. A modification of the hand method is shown below, implemented using some of the routines already described. The times were calculated using the corresponding arguments. The secret of the shorter calculatio time reside in the special shift routine kkmSHRRP. This routine will shift right only a part of a number using **rt7** as index for this shift. It is implemented as two parts whether the shift occurs in the whole number or only on the right most long. For easier handling the significant is scaled by a factor of 5, making the multiplication by 20 (see hand algorithm at Wikipedia) unnecessary. The exponent is calculated using a simple dive by two in binary, because it is stored as a two's complement number.

```
' Calculates the square root of the argument in A
' As it is it takes new one, (old one was without self-modifying code)
:
'
' Input    cycles  cycles   result
'          old one new one
'      .78 158344  111192   0.883176086632
'     1.0    7012    6564   1.0
'     2.0  107940   76332   1.41421356237
'     5.0  169028  118560   2.23606797749
'    50.0  142408  100176   7.07106781186
'   100.0    7012    6564  10.0
'  1000.0  129128   90996   31.6227766016
' 1.3e+51  134440   94668   3.60555127546e+25
ERR_SQRN = 3

kkBCDSQR        call    #kkmCMPAZ
        if_z    call    #LOADZTOR
        if_z    jmp     #kkBCDSQR_ret       ' argument is zero

                cmp     rASgn,#0   wz
        if_nz   mov     rerr,#ERR_SQRN      ' argument is negative
        if_nz   jmp     #kkBCDSQR_ret

                movs    kkmADD8_1,#rA1
```

```
                movs    kkmADD8_2,#rA1          ' B+B
                call    #kkmADDSM
                mov     rB1,rt6
                mov     rB,rt5                ' B=A+A
                movs    kkmADD8_1,#rB1
                movs    kkmADD8_2,#rB1          ' B=4*A
                call    #kkmADDSM
                mov     rB1,rt6
                mov     rB,rt5
                movs    kkmADD8_1,#rB1
                movs    kkmADD8_2,#rA1          ' A=A+B
                call    #kkmADDSM               ' A=5*A
                mov     rA1,rt6
                mov     rA,rt5

                mov     rRExp,rAExp
                test    rAExp,#1   wz
        if_z    call    #kkmSHRA15              ' shift right if exponen
t was even

                mov     rR,cnt_FIVE
                mov     rR1,#0                ' rt6:rt7 is used to calcu
late the digits
                mov     rB,cnt_ONE
                mov     rB1,#0                ' we initialize constant
                mov     rt7,#12               ' 12 digits
kkBCDSQR_10     call    #kkmSHRRP
kkBCDSQR_17     cmp     rA,rR   wz wc
        if_z    cmp     rA1,rR1   wz wc

        if_c    jmp     #kkBCDSQR_20
' subtracts result
                movs    kkmSUB8_1,#rA1
                movs    kkmSUB8_2,#rR1
                call    #kkmSUBSM
                mov     rA1,rt6
                mov     rA,rt5
' adds one to result
                movs    kkmADD8_1,#rR1
                movs    kkmADD8_2,#rB1
                call    #kkmADDSM
                mov     rR1,rt6
                mov     rR,rt5
                jmp     #kkBCDSQR_17

kkBCDSQR_20     call    #kkmSHLA15              ' shifts left remainde
```

```
r
                call    #kkmSHRB15              ' shift right constant
kkBCDSQR_25     djnz    rt7,#kkBCDSQR_10

kkBCDSQR_ret    ret

' Rotate right with mask in rt7
kkmSHRRP        mov     rt4,cnt_ff              ' 0xffff_ffff
                mov     rt3,rt7                 ' shift count
                cmpsub  rt3,#5   wc wr
                shl     rt3,#2
                shl     rt4,rt3                 ' prepares mask
        if_c    jmp     #kkmSHRRP_20              ' we will see

                mov     rt2,rR1
                andn    rt2,rt4
                shr     rt2,#4
                and     rR1,rt4
                or      rR1,rt2
                jmp     #kkmSHRRP_ret

kkmSHRRP_20     shr     rR1,#4
                mov     rt2,rR
                shl     rt2,#28
                or      rR1,rt2                 ' lower long ready
                mov     rt2,rR
                andn    rt2,rt4                 ' right half of high wor
d ready
                shr     rt2,#4
                and     rR,rt4
                or      rR,rt2
kkmSHRRP_ret    ret
```

Transcendentals

The need for floating point can be somewhat mitigated using for instance Fixed notation (a variant for binary floating point), but when transcendental functions are needed, it could be difficult to avoid its use. First of all we will consider several methods to implement some of the functions, angle functions, power and logarithm. With some basic identities, all possible functions can be obtained.

The first function to consider will be the sine function. As everyone knows there are a number of methods to calculate it, power series of several kinds (all related to the Taylor or McLaurin series) and the über-toll CORDIC methods. Floating point coprocessors calculate them using power series while pocket calculators on the other hand, (the HP series and TI series for example) use the CORDIC method, because they work with decimal numbers (BCD) not with binary numbers.

A Taylor series for the sine will be:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + E$$

The error term will be the difference between the real value (with an infinite number of terms) and the one calculated with a reasonable number of terms.
If 16 exact digits are to be calculated a 23 term series is needed. This means, storing 23 constants and doing 22 additions and 24 multiplications, using a simplified version (without Error term), for 4 terms there are 5 multiplications and 3 additions/subtractions:

$$\sin(x) = x * \left(1 - x^2 * \left(\frac{1}{3!} + x^2 * \left(\frac{1}{5!} - \frac{x^2}{7!}\right)\right)\right)$$

When fast multiplication is available, this method could be used. In real life, some other considerations are taken, like angle reduction (using only 1 quadrant) and partial approximations using a table of pre-calculated constants.

The CORDIC method was developed, sadly, as an aid in missile guiding systems (read: to kill people). But despite that, some good came from it, as is it in widespread use for more edifying purposes, hopefully. It is based on an old method developed in the 17th century by mathematician Briggs. This method is based on algebraic functions, and can be thought out as an I infer the result working with the argument. There is no direct correlation between the argument and the result, because firstly an intermediate set of results has to be calculated to be used to calculate a result. (See Jacques Laporte's site)

Lets see it with a sine case:

```
Soon to come
```

Note 1: This examples can be found in this file here ready to be tested with pPropellerSim or if you change the extension and add a wrapper... a ready object for you to test on a propeller. Note the license (LGPL v2)!

Note 2: The double precision package is here.

Method calls (subroutine and function calls) in Spin are a little complicated from a bytecode and run-time execution perspective.

Rather than have a 'call' to a specific address where the method's executable code is, as is used in PASM and other programming languages, Spin creates a 'method pointer table' which holds the addresses of the methods, and a method call is done through that table using an index of the number of the method to call. This is a technique which was used with the Parallax Basic Stamp and may be the basis for the decision to use the same technique in Spin, however the reasoning behind the choice of architecture is not known.

The method table is placed at the start of the main program or object it relates to and is a list of 32-bit longs, each entry split into two 16-bit words. The first long is at index +0, the second at +1 and so on.

The first entry (+0) is used to link all objects in an application together and is not discussed further in this document. The full puropose of this entry and how it has been used by the Spin interpreter has not yet been determined.

The second entry (+1) is the one relating to the first or only PUB method defined in the program or object. All PUB method entries are placed in the 'method pointer table' followed by any PRI method entries if any exist, and finally come entries to any sub-objects that may have been included.

A call to a method is a call via an index into the 'method pointer table'. A call to the first PUB method of the program would see bytecode generated in the form of "CALL +1", a call to a second PUB method if it existed (or first PRI method if it did not) would be a bytecode sequence of "CALL +2". In each case the Spin Interpreter will locate the 'method pointer table', find the second (+1) or third (+2) entry respectively and use the entry to invoke the method called.

The two words which make up the related entry in the 'method pointer table' are, frstly, the actual address of the method's code to execute, and the second a number of bytes by which to increase the stack pointer when the call is made. This allows space to be allocated for local variables used within the called method. As local variables are all 32-bit longs, the number to increase the stack pointer by will be zero (no local variables) or a multiple of four bytes. Note that the stack pointer is incremented but there is no clearing of the data on the stack. This is why local variables are not initialised when a method is entered.

Actually making a call to a method is a little more complicated and is a three stage affair. Firstly the 'call stack framing information' is created; this determines what should be done when the call returns, whether any value returned from the method should be discarded or left on the stack and whether an abort from within the method should be caught or allowed to fall through.

Once the 'call stack framing information' has been determined, any paramaters to pass to the routine are evaluated and the resultsof each evaluation is left on the stack. Finally, the actual call to the method is made. At the point the method is entered it should be noted that the passed-in parameter values are stored on the stack immediately preceding any local variables used. Within the method itself both parameters and local variables are treated the same by the Spin Interpreter, the Spin Compiler of the Propeller Tool having determined where each will respectively be on the stack.

When the method completes, reaches the end of its code or encounters a 'return' or 'abort', the Spin

Interpreter unravels the stack, handles the returned value and deals with any abort processing necessary. The Spin Interpreter then continues executing the bytecode after the method call.

Note that all methods return a value which can be one of three things; a return value specified by a 'return' statement, an abort value specified by an 'abort' statement, or whatever value is held in the 'result' variable when the end of method code, or a 'return' or 'abort' statement with no expression is encountered. The 'result' variable exists for all methods and can be considered as the first local variable used within the method, before any parameters and any explicitly defined local variables. Unlike explicit local variables, the 'result' variable is always initialised to zero when a method is called.

Function Pointers

It has been noted that the Spin Language does not support 'function pointers', for example the address of a 'PRI GetCounter' cannot be obtained by using an '@GetCounter' expression even though its address is known at both compile time and run-time.

The lack of function pointers seems inexplicable to those expecting a call to a function to be simply implemented as a call to the function's executable code, but because calls do not make sense with the Spin Bytecode architecture a function pointer could not be used in the same way; a 'function pointer' in Spin would need to be an index into the 'method pointer table' and not the address of the method's executable code.

Spin methods do not include a mechanism within themselves for making local stack space available (the necessary information for that is held in the 'method pointer table') so even if the address of a method's executable code could be found, it could not be reliably executed unless it used no local variables. Because the stack pointer would not have been adjusted to take account of those local variables, any executing code would interfere with the stack space those local variables would effectively be sharing at that time leading to all manner of peculiar results.

Because the Spin architecture was not designed to allow methods to be called directly it makes no real sense in most cases to provide any means to obtain the address of a method's executable code and there is no easy way to use function pointers as there is in other languages.

This is also the reason that there are no simple 'call-back mechanisms' because they intrinsically rely upon function pointers to work.

Unfortunately the Spin language provides no known mechanism to identify the index of a particular method in the 'method pointer table' so there is no easy mechanism to call particular methods programmatically at run-time nor to provide call-back mechanisms. Dynamic selection of methods to call at run-time can only really be done using 'if-else' or 'case' statements.

It is possible to 'fix-up' or 'patch' the bytecode at run-time (either in-line code or the 'method pointer table') to dynamically select a method to call, but this is no simple task and also relies on having to manually determine the indexing of the methods being called which can easily and frequently change as

code is developed. It also requires considerable understanding of teh Spin bytecode itself. It is therefore complicated and potentially error prone to use such a strategy, and the mechanism needs additional protections to allow it to be used with re-enterent code which is shared by one or more parallel operating Cog programs.

The Future

There is no reason that the Spin Language could not provide a means of obtaining an index for a method into the 'method pointer table' nor not provide a means to call a method using that information.

There has been no indication that either ability will be provided by Parallax but it is something which could potentially be provided for by a third-party Spin Compiler.

Inter-Object Method Calls

Inter-object method calls, and consequently 'call back methods', have an additional level of complication requiring additional 'Object Base' and 'Variable Base' pointers to be updated along with the stack pointer when a call is made.

This is all handled by the Spin Interpreter again using information held in the 'method pointer table'. The bytecode for a call into another object's method is in the form of 'CALLOBJ +N,+M' where the "+N" is an index into the 'method pointer table' as with a normal method call but the entry held there is different.

The first word of such an entry is the 'Object Base' of the object referenced and the second word is an amount of bytes to update the 'Variable Base' by when that object's methods are called. Updating the 'Variable Base' is what allows each object to have their own entirely independant VAR sections, and the VAR sections of each object instantiation to be be unique to that object.

The first step of an inter-object method call is therefore to update the 'Object Base' and 'Variable Base' pointers and thereafter the '+M' reference to the method to be called relates to the 'method pointer table' of the object referenced and can be handled just as a method call within the same object can be.

For anyone undertaking the development of a Spin Interpreter, it must be noted that the alteration of the 'Object Pointer' and 'Variable Base' must be done at the last instant, just as control is passed to the called method. Changes cannot be made to either at the time framing information is calculated as this is before parameters for the call is evaluated and both need to remain correct for those evaluatons to take place.

When an inter-object call completes, the Spin Interpreter again unravals the stack as for a call within an object and restores the 'Object Base' and 'Variable Base' pointers as they need to be for the object returned to. To simplify handling of method returns, the framing information pushed to the stack appears to contains the same informaton regardless of whether a method call is inter-object or within the same object. A return from a same-object method call will simply 'restore' both 'Object base' and 'Variable

Base' to what they were.

This means more framing information is pushed to the stack than is absolutely necessary on a same-object method call but it does mean that method return handling is common regardless of how the call was made and the stack frame will be the same when seen from within a method regardless of how it were called for a method which is able to use information pushed onto the stack before it was called.

Driving a Controller-less Monochrome LCD

The powerful video circuitry inside the propeller can be very useful for driving TVs or Monitors. But to drive a controllerless monochrome LCD panel when multiple data bits have to be transferred at once one has to recur to a more normal procedure. This note will walk you on how to achieve this little task.

This circuit is an extension to pPropQL020 to add a LCD terminal. The expansion port matches that on pPropQL020.

A typical dual-scan LCD

Mono LCD panels come in two basic flavours, single scan and dual scan. Single scan means that one the displayed frame is transferred one line at a time. In a dual scan panel two lines are transferred at once. This reduces frame time but complicates the driving because two areas of memory have to be read for a group of pixels unless the data has been stored interleaved.

The chosen panel is a KL6448. This panel has a 640x480 resolution and has a 144x109 mm of viewing area. The input signals have to be 5V CMOS and the LCD bias voltage is of +24 V.

Pin	Description
1	FRAME - this signals marks the start of a frame
2	LOAD - this signal loads a line into the output drivers
3	CP - the data is sampled on the falling edge of this clock signal
4	DISPON - A high level indicates that the display shoud be active
5	+5V - Supply voltage
6	GND - Ground
7	VEE - Positive bias voltage around 24 V, only activate with DISPON
8	UD0 Upper half data

9	UD1
10	UD2
11	UD3
12	LD0 Lower half data
13	LD1
14	LD2
15	LD3

The interface is pretty simple, and so is the logic to drive it. A circuit is shown below.

According to the datasheet this display needs between 0.8*VCC and VCC as high-level, so I used a couple of HCT gates and a 245 as buffers/level-shifters. It is not really important which one, only that it has xxT inputs, TTL compatible because the propeller will put some 3 to 3.3 V as output. A HC gate could probably work too.

Driving the panel

The control signals, **FRM**, **LOAD** and **CP** have to be driven in a timely manner. **FRM** will rise to indicate the start of a frame and will fall after **LOAD** has fallen. **LOAD** will rise and fall after 80 pairs of data have been shifted in. **CP** will be used to shift the data in for every high-to-low transition. An image is worth a thousand words so here it is:

With that image in mind, I present some code to drive this panel. The code uses 3 cogs, yes 3. One is used to generate all control signals, and the other two shift the data. It could probably be done with 2 cogs but it is a nice example of synchronised work. To sync a **waitcnt** instruction is used with a future value of **CNT**+80000 cycles (10 ms).
The presented code will display a 80x30 text image. The variable screen is used to hold the data to display. A 8x16 font normally used in VGA cards is also included. This allows to show the text.

Every frame is composed of 240 lines. Each line corresponds to one of the text lines and is inside a font line. So each frame is subsequently divided into chunks of 16 lines, 15 of them. The lack of 38 kbytes of

memory reduces the graphic capabilities for such a big display but some alternatives like 640x400 or 640x240 or 320x480 or maybe 512x480 can be used.
To change from text to graphics on-the-fly a HUB variable is monitored every frame. When this variable is for instance "GRP0" a 640x400 image is generated. Other values could also be used for other modes. Returning to text mode means that the font has to be loaded again from either EEPROM or from the host. that should allow for custom chars, if needed.

DUal scan LCD, the two halves show the same data

```
CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

{
LCD Terminal - A terminal bundle for a 640x480 monochrome LCD

(c) 2009 R. A. Paz Schmidt - Pacito.Sys - hppacito <at> gmail dot com
}

VAR
  byte screen[80*30]

PUB start
  cognew(@LCD_CTRL, @screen) ' control COG
  cognew(@LCD_COGH, @screen) ' higher display part
  cognew(@LCD_COGL, @screen) ' lower display part
  'cognew(@IO_COG, 0) ' I/O cog

CON

' Display control signals

k_FRM = 31      ' Frame signal, signals frame start, aka vertical sync
k_LOAD = 30     ' Line load, signals end of line shift, horizontal syn
c
k_CLK = 25      ' Data Clock, data is loaded in the falling edge
k_DISPON = 24   ' Display ON, activates the display and the LCD bias s
upply (+24 V)

k_HDISPDATA = $00_f0_00_00  ' Upper half data lines
k_LDISPDATA = $00_0f_00_00  ' Lower half data lines
```

```
k_HDISPSHL  = 16
k_LDISPSHL  = 12
DAT

              byte      "Hallo Welt !! Pacito.Sys !!"



DAT
              org       0

LCD_CTRL      mov       OUTA, #0
              mov       DIRA, v0_k_DIRA         ' sets outputs

              mov       v0_r0, CNT
              add       v0_r0, v0_k_delay
              wrlong    v0_r0, v0_k_sync
              waitcnt   v0_r0, #0

c0_t_frame    or        OUTA, v0_k_FRM          ' asserts FRM

' 15 font lines
              mov       v0_lcnt, #15
c0_t_15line   mov       v0_r3, #16              ' 15 chars per half sc
reen
' a font line is a group of 16 vert lines
c0_t_fontline mov       v0_r2, #80              ' 80 chars per line
c0_t_line     rdbyte    v0_r0, v0_k_sync        ' uses HUB
              nop
              nop

              rdbyte    v0_r0, v0_k_sync        ' uses HUB
              nop
              or        OUTA, v0_k_CLK          ' slot for OUT
              andn      OUTA, v0_k_CLK
              nop
              or        OUTA, v0_k_CLK          ' slot for OUT
              andn      OUTA, v0_k_CLK

              djnz      v0_r2, #c0_t_line

              or        OUTA, v0_k_LOAD
              nop
              nop
              andn      OUTA, v0_k_LOAD
```

```
              nop
              nop
              andn      OUTA, v0_k_FRM
              djnz      v0_r3, #c0_t_fontline

              nop
              nop
              djnz      v0_lcnt, #c0_t_15line
              or        OUTA, v0_k_DISPON       ' turns ON display aft
er 1 frame
              jmp       #c0_t_frame



v0_lcnt       long      0       ' line counter
v0_r0         long      0
v0_r1         long      0
v0_r2         long      0
v0_r3         long      0
c0_dispon     long      0       ' set to 1 when 1 frame at least was s
ent, indicates that the DISP = ON

v0_k_DIRA     long      (1<<k_FRM)|(1<<k_LOAD)|(1<<k_CLK)|(1<<k_DISPON
)
v0_k_FRM      long      1<<k_FRM
v0_k_LOAD     long      1<<k_LOAD
v0_k_CLK      long      1<<k_CLK
v0_k_DISPON   long      1<<k_DISPON
v0_k_HDTA     long      k_HDISPDATA
v0_k_delay    long      80_000
v0_k_sync     long      $7ffc

              fit       $1f0
DAT
              org       0

LCD_COGH      mov       OUTA, #0
              mov       DIRA, v1_k_HDTA         ' sets to outputs

' There are two modes of operation:
' 640x480 text with 80x30 chars
' and
' 640x400 graphics, centered with 40 lines above and below
'
' We start in text mode
'
```

```
              rdlong    v1_r0, v1_k_sync
              waitcnt   v1_r0, #0

c1_t_frame    mov       v1_k_textbuff, PAR      ' gets text pointer

' 15 font lines
              mov       v1_lcnt, #15            ' 15 chars per half sc
reen
c1_t_15line   mov       v1_r3, #16              ' chars are 16 lines h
igh
' a font line is a group of 16 vert lines
c1_t_fontline mov       v1_r2, #80              ' 80 chars per line
c1_t_line     rdbyte    v1_r0, v1_k_textbuff
              shl       v1_r0, #4
              add       v1_r0, v1_k_font

              rdbyte    OUTA, v1_r0             ' reads font data
              shl       OUTA, #k_HDISPSHL       ' shifts data
              add       v1_k_textbuff, #1       ' increments pointer
              shl       OUTA, #4                ' shifts lower nibble,
 next 4 pixels
              nop
              nop
              nop

              djnz      v1_r2, #c1_t_line

              sub       v1_k_textbuff, #80
              add       v1_k_font, #1           ' increments font poin
ter to get next font line
              nop                               ' delay between groups
 of lines
              nop
              nop
              nop
              nop

              djnz      v1_r3, #c1_t_fontline

              sub       v1_k_font, #16          ' restores font pointe
r
              add       v1_k_textbuff, #80      ' increments line poin
ter
              djnz      v1_lcnt, #c1_t_15line
              nop
              jmp       #c1_t_frame
```

```
v1_lcnt       long      0       ' line counter
v1_r0         long      0
v1_r1         long      0
v1_r2         long      0
v1_r3         long      0
v1_k_font     long      @@@VGAFONT16
v1_k_textbuff long      0
v1_k_DIRA     long      0
v1_k_HDTA     long      k_HDISPDATA
v1_k_delay    long      40_000
v1_k_sync     long      $7ffc

              fit       $1f0
DAT

              org       0
LCD_COGL      mov       OUTA, #0
              mov       DIRA, v2_k_LDTA         ' sets to outputs
              add       v2_k_ibuffptr, PAR

' There are two modes of operation:
' 640x480 text with 80x30 chars
' and
' 640x400 graphics, centered with 40 lines above and below
'
' We start in text mode
'
              rdlong    v2_r0, v2_k_sync
              waitcnt   v2_r0, #0

c2_t_frame    mov       v2_k_textbuff, v2_k_ibuffptr     ' gets text p
ointer

' 15 font lines
              mov       v2_lcnt, #15            ' 15 chars per half sc
reen
c2_t_15line   mov       v2_r3, #16              ' chars are 16 lines h
igh
' a font line is a group of 16 vert lines
c2_t_fontline mov       v2_r2, #80              ' 80 chars per line
c2_t_line     rdbyte    v2_r0, v2_k_textbuff
              shl       v2_r0, #4
              add       v2_r0, v2_k_font

              rdbyte    OUTA, v2_r0             ' reads font data
```

```
              shl       OUTA, #k_LDISPSHL       ' shifts data
              nop
              shl       OUTA, #4                ' shifts lower nibble,
 next 4 pixels
              nop
              add       v2_k_textbuff, #1       ' increments pointer
              nop
              djnz      v2_r2, #c2_t_line

              sub       v2_k_textbuff, #80      ' reverts to the same
text line
              add       v2_k_font, #1           ' increments font poin
ter to get next font line
              nop
              nop
              nop
              nop
              nop

              djnz      v2_r3, #c2_t_fontline

              sub       v2_k_font, #16          ' restores font pointe
r
              add       v2_k_textbuff, #80      ' increments line poin
ter
              djnz      v2_lcnt, #c2_t_15line

              nop
              jmp       #c2_t_frame

v2_lcnt       long      0       ' line counter
v2_r0         long      0
v2_r1         long      0
v2_r2         long      0
v2_r3         long      0
v2_k_font     long      @@@VGAFONT16
v2_k_textbuff long      0
v2_k_DIRA     long      0
v2_k_LDTA     long      k_LDISPDATA
v2_k_ibuffptr long      15*80
v2_k_sync     long      $7ffc

              fit       $1f0

DAT
' * These fonts come from ftp://ftp.simtel.net/pub/simtelnet/msdos/scr
```

```
een/fntcol16.zip
' * The package is (c) by Joseph Gil
' * The individual fonts are public domain

VGAFONT16      byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
               byte $00, $00, $7e, $81, $a5, $81, $81, $bd, $99, $81,
$81, $7e, $00, $00, $00, $00
               byte $00, $00, $7e, $ff, $db, $ff, $ff, $c3, $e7, $ff,
$ff, $7e, $00, $00, $00, $00
               byte $00, $00, $00, $00, $6c, $fe, $fe, $fe, $fe, $7c,
$38, $10, $00, $00, $00, $00
               byte $00, $00, $00, $00, $10, $38, $7c, $fe, $7c, $38,
$10, $00, $00, $00, $00, $00
               byte $00, $00, $00, $18, $3c, $3c, $e7, $e7, $e7, $18,
$18, $3c, $00, $00, $00, $00
               byte $00, $00, $00, $18, $3c, $7e, $ff, $ff, $7e, $18,
$18, $3c, $00, $00, $00, $00
               byte $00, $00, $00, $00, $00, $00, $18, $3c, $3c, $18,
$00, $00, $00, $00, $00, $00
               byte $ff, $ff, $ff, $ff, $ff, $ff, $e7, $c3, $c3, $e7,
$ff, $ff, $ff, $ff, $ff, $ff
               byte $00, $00, $00, $00, $00, $3c, $66, $42, $42, $66,
$3c, $00, $00, $00, $00, $00
               byte $ff, $ff, $ff, $ff, $ff, $c3, $99, $bd, $bd, $99,
$c3, $ff, $ff, $ff, $ff, $ff
               byte $00, $00, $1e, $0e, $1a, $32, $78, $cc, $cc, $cc,
$cc, $78, $00, $00, $00, $00
               byte $00, $00, $3c, $66, $66, $66, $66, $3c, $18, $7e,
$18, $18, $00, $00, $00, $00
               byte $00, $00, $3f, $33, $3f, $30, $30, $30, $30, $70,
$f0, $e0, $00, $00, $00, $00
               byte $00, $00, $7f, $63, $7f, $63, $63, $63, $63, $67,
$e7, $e6, $c0, $00, $00, $00
               byte $00, $00, $00, $18, $18, $db, $3c, $e7, $3c, $db,
$18, $18, $00, $00, $00, $00
               byte $00, $80, $c0, $e0, $f0, $f8, $fe, $f8, $f0, $e0,
$c0, $80, $00, $00, $00, $00
               byte $00, $02, $06, $0e, $1e, $3e, $fe, $3e, $1e, $0e,
$06, $02, $00, $00, $00, $00
               byte $00, $00, $18, $3c, $7e, $18, $18, $18, $7e, $3c,
$18, $00, $00, $00, $00, $00
               byte $00, $00, $66, $66, $66, $66, $66, $66, $66, $00,
$66, $66, $00, $00, $00, $00
               byte $00, $00, $7f, $db, $db, $db, $7b, $1b, $1b, $1b,
$1b, $1b, $00, $00, $00, $00
```

```
                byte $00, $7c, $c6, $60, $38, $6c, $c6, $c6, $6c, $38,
$0c, $c6, $7c, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $00, $fe, $fe,
$fe, $fe, $00, $00, $00, $00
                byte $00, $00, $18, $3c, $7e, $18, $18, $18, $7e, $3c,
$18, $7e, $00, $00, $00, $00
                byte $00, $00, $18, $3c, $7e, $18, $18, $18, $18, $18,
$18, $18, $00, $00, $00, $00
                byte $00, $00, $18, $18, $18, $18, $18, $18, $18, $7e,
$3c, $18, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $18, $0c, $fe, $0c, $18,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $30, $60, $fe, $60, $30,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $c0, $c0, $c0, $fe,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $24, $66, $ff, $66, $24,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $10, $38, $38, $7c, $7c, $fe,
$fe, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $fe, $fe, $7c, $7c, $38, $38,
$10, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $18, $3c, $3c, $3c, $18, $18, $18, $00,
$18, $18, $00, $00, $00, $00
                byte $00, $66, $66, $66, $24, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $6c, $6c, $fe, $6c, $6c, $6c, $fe,
$6c, $6c, $00, $00, $00, $00
                byte $18, $18, $7c, $c6, $c2, $c0, $7c, $06, $06, $86,
$c6, $7c, $18, $18, $00, $00
                byte $00, $00, $00, $00, $c2, $c6, $0c, $18, $30, $60,
$c6, $86, $00, $00, $00, $00
                byte $00, $00, $38, $6c, $6c, $38, $76, $dc, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $30, $30, $30, $60, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $0c, $18, $30, $30, $30, $30, $30, $30,
$18, $0c, $00, $00, $00, $00
                byte $00, $00, $30, $18, $0c, $0c, $0c, $0c, $0c, $0c,
$18, $30, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $66, $3c, $ff, $3c, $66,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $18, $18, $7e, $18, $18,
$00, $00, $00, $00, $00, $00
```

```
                byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $18,
$18, $18, $30, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $fe, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
$18, $18, $00, $00, $00, $00
                byte $00, $00, $00, $00, $02, $06, $0c, $18, $30, $60,
$c0, $80, $00, $00, $00, $00
                byte $00, $00, $3c, $66, $c3, $c3, $db, $db, $c3, $c3,
$66, $3c, $00, $00, $00, $00
                byte $00, $00, $18, $38, $78, $18, $18, $18, $18, $18,
$18, $7e, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $06, $0c, $18, $30, $60, $c0,
$c6, $fe, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $06, $06, $3c, $06, $06, $06,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $0c, $1c, $3c, $6c, $cc, $fe, $0c, $0c,
$0c, $1e, $00, $00, $00, $00
                byte $00, $00, $fe, $c0, $c0, $c0, $fc, $06, $06, $06,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $38, $60, $c0, $c0, $fc, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $fe, $c6, $06, $06, $0c, $18, $30, $30,
$30, $30, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $c6, $c6, $7c, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $c6, $c6, $7e, $06, $06, $06,
$0c, $78, $00, $00, $00, $00
                byte $00, $00, $00, $00, $18, $18, $00, $00, $00, $18,
$18, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $18, $18, $00, $00, $00, $18,
$18, $30, $00, $00, $00, $00
                byte $00, $00, $00, $06, $0c, $18, $30, $60, $30, $18,
$0c, $06, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $7e, $00, $00, $7e, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $60, $30, $18, $0c, $06, $0c, $18,
$30, $60, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $c6, $0c, $18, $18, $18, $00,
$18, $18, $00, $00, $00, $00
                byte $00, $00, $00, $7c, $c6, $c6, $de, $de, $de, $dc,
$c0, $7c, $00, $00, $00, $00
                byte $00, $00, $10, $38, $6c, $c6, $c6, $fe, $c6, $c6,
$c6, $c6, $00, $00, $00, $00
                byte $00, $00, $fc, $66, $66, $66, $7c, $66, $66, $66,
$66, $fc, $00, $00, $00, $00
```

```
                byte $00, $00, $3c, $66, $c2, $c0, $c0, $c0, $c0, $c2,
$66, $3c, $00, $00, $00, $00
                byte $00, $00, $f8, $6c, $66, $66, $66, $66, $66, $66,
$6c, $f8, $00, $00, $00, $00
                byte $00, $00, $fe, $66, $62, $68, $78, $68, $60, $62,
$66, $fe, $00, $00, $00, $00
                byte $00, $00, $fe, $66, $62, $68, $78, $68, $60, $60,
$60, $f0, $00, $00, $00, $00
                byte $00, $00, $3c, $66, $c2, $c0, $c0, $de, $c6, $c6,
$66, $3a, $00, $00, $00, $00
                byte $00, $00, $c6, $c6, $c6, $c6, $fe, $c6, $c6, $c6,
$c6, $c6, $00, $00, $00, $00
                byte $00, $00, $3c, $18, $18, $18, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $00, $1e, $0c, $0c, $0c, $0c, $0c, $cc, $cc,
$cc, $78, $00, $00, $00, $00
                byte $00, $00, $e6, $66, $66, $6c, $78, $78, $6c, $66,
$66, $e6, $00, $00, $00, $00
                byte $00, $00, $f0, $60, $60, $60, $60, $60, $60, $62,
$66, $fe, $00, $00, $00, $00
                byte $00, $00, $c3, $e7, $ff, $ff, $db, $c3, $c3, $c3,
$c3, $c3, $00, $00, $00, $00
                byte $00, $00, $c6, $e6, $f6, $fe, $de, $ce, $c6, $c6,
$c6, $c6, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $c6, $c6, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $fc, $66, $66, $66, $7c, $60, $60, $60,
$60, $f0, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $c6, $c6, $c6, $c6, $c6, $d6,
$de, $7c, $0c, $0e, $00, $00
                byte $00, $00, $fc, $66, $66, $66, $7c, $6c, $66, $66,
$66, $e6, $00, $00, $00, $00
                byte $00, $00, $7c, $c6, $c6, $60, $38, $0c, $06, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $ff, $db, $99, $18, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $00, $c6, $c6, $c6, $c6, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $c3, $c3, $c3, $c3, $c3, $c3, $c3, $66,
$3c, $18, $00, $00, $00, $00
                byte $00, $00, $c3, $c3, $c3, $c3, $c3, $db, $db, $ff,
$66, $66, $00, $00, $00, $00
                byte $00, $00, $c3, $c3, $66, $3c, $18, $18, $3c, $66,
$c3, $c3, $00, $00, $00, $00
                byte $00, $00, $c3, $c3, $c3, $66, $3c, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
```

```
                byte $00, $00, $ff, $c3, $86, $0c, $18, $30, $60, $c1,
$c3, $ff, $00, $00, $00, $00
                byte $00, $00, $3c, $30, $30, $30, $30, $30, $30, $30,
$30, $3c, $00, $00, $00, $00
                byte $00, $00, $00, $80, $c0, $e0, $70, $38, $1c, $0e,
$06, $02, $00, $00, $00, $00
                byte $00, $00, $3c, $0c, $0c, $0c, $0c, $0c, $0c, $0c,
$0c, $3c, $00, $00, $00, $00
                byte $10, $38, $6c, $c6, $00, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
$00, $00, $00, $ff, $00, $00
                byte $30, $30, $18, $00, $00, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $78, $0c, $7c, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $00, $e0, $60, $60, $78, $6c, $66, $66, $66,
$66, $7c, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $7c, $c6, $c0, $c0, $c0,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $1c, $0c, $0c, $3c, $6c, $cc, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $7c, $c6, $fe, $c0, $c0,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $38, $6c, $64, $60, $f0, $60, $60, $60,
$60, $f0, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $76, $cc, $cc, $cc, $cc,
$cc, $7c, $0c, $cc, $78, $00
                byte $00, $00, $e0, $60, $60, $6c, $76, $66, $66, $66,
$66, $e6, $00, $00, $00, $00
                byte $00, $00, $18, $18, $00, $38, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $00, $06, $06, $00, $0e, $06, $06, $06, $06,
$06, $06, $66, $66, $3c, $00
                byte $00, $00, $e0, $60, $60, $66, $6c, $78, $78, $6c,
$66, $e6, $00, $00, $00, $00
                byte $00, $00, $38, $18, $18, $18, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $e6, $ff, $db, $db, $db,
$db, $db, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $dc, $66, $66, $66, $66,
$66, $66, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $7c, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $dc, $66, $66, $66, $66,
$66, $7c, $60, $60, $f0, $00
```

```
                byte $00, $00, $00, $00, $00, $76, $cc, $cc, $cc, $cc,
$cc, $7c, $0c, $0c, $1e, $00
                byte $00, $00, $00, $00, $00, $dc, $76, $66, $60, $60,
$60, $f0, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $7c, $c6, $60, $38, $0c,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $10, $30, $30, $fc, $30, $30, $30, $30,
$36, $1c, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $cc, $cc, $cc, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $c3, $c3, $c3, $c3, $66,
$3c, $18, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $c3, $c3, $c3, $db, $db,
$ff, $66, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $c3, $66, $3c, $18, $3c,
$66, $c3, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $c6, $c6, $c6, $c6, $c6,
$c6, $7e, $06, $0c, $f8, $00
                byte $00, $00, $00, $00, $00, $fe, $cc, $18, $30, $60,
$c6, $fe, $00, $00, $00, $00
                byte $00, $00, $0e, $18, $18, $18, $70, $18, $18, $18,
$18, $0e, $00, $00, $00, $00
                byte $00, $00, $18, $18, $18, $18, $00, $18, $18, $18,
$18, $18, $00, $00, $00, $00
                byte $00, $00, $70, $18, $18, $18, $0e, $18, $18, $18,
$18, $70, $00, $00, $00, $00
                byte $00, $00, $76, $dc, $00, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $10, $38, $6c, $c6, $c6, $c6,
$fe, $00, $00, $00, $00, $00
                byte $00, $00, $3c, $66, $c2, $c0, $c0, $c0, $c2, $66,
$3c, $0c, $06, $7c, $00, $00
                byte $00, $00, $cc, $00, $00, $cc, $cc, $cc, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $0c, $18, $30, $00, $7c, $c6, $fe, $c0, $c0,
$c6, $7c, $00, $00, $00, $00
                byte $00, $10, $38, $6c, $00, $78, $0c, $7c, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $00, $cc, $00, $00, $78, $0c, $7c, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $60, $30, $18, $00, $78, $0c, $7c, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $38, $6c, $38, $00, $78, $0c, $7c, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $00, $00, $00, $3c, $66, $60, $60, $66, $3c,
$0c, $06, $3c, $00, $00, $00
```

```
                byte $00, $10, $38, $6c, $00, $7c, $c6, $fe, $c0, $c0,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $c6, $00, $00, $7c, $c6, $fe, $c0, $c0,
$c6, $7c, $00, $00, $00, $00
                byte $00, $60, $30, $18, $00, $7c, $c6, $fe, $c0, $c0,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $66, $00, $00, $38, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $18, $3c, $66, $00, $38, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $60, $30, $18, $00, $38, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $c6, $00, $10, $38, $6c, $c6, $c6, $fe, $c6,
$c6, $c6, $00, $00, $00, $00
                byte $38, $6c, $38, $00, $38, $6c, $c6, $c6, $fe, $c6,
$c6, $c6, $00, $00, $00, $00
                byte $18, $30, $60, $00, $fe, $66, $60, $7c, $60, $60,
$66, $fe, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $6e, $3b, $1b, $7e, $d8,
$dc, $77, $00, $00, $00, $00
                byte $00, $00, $3e, $6c, $cc, $cc, $fe, $cc, $cc, $cc,
$cc, $ce, $00, $00, $00, $00
                byte $00, $10, $38, $6c, $00, $7c, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $c6, $00, $00, $7c, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $60, $30, $18, $00, $7c, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $30, $78, $cc, $00, $cc, $cc, $cc, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $60, $30, $18, $00, $cc, $cc, $cc, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $00, $c6, $00, $00, $c6, $c6, $c6, $c6, $c6,
$c6, $7e, $06, $0c, $78, $00
                byte $00, $c6, $00, $7c, $c6, $c6, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $c6, $00, $c6, $c6, $c6, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $18, $18, $7e, $c3, $c0, $c0, $c3, $7e,
$18, $18, $00, $00, $00, $00
                byte $00, $38, $6c, $64, $60, $f0, $60, $60, $60, $60,
$e6, $fc, $00, $00, $00, $00
                byte $00, $00, $c3, $66, $3c, $18, $ff, $18, $ff, $18,
$18, $18, $00, $00, $00, $00
                byte $00, $fc, $66, $66, $7c, $62, $66, $6f, $66, $66,
$66, $f3, $00, $00, $00, $00
```

```
                byte $00, $0e, $1b, $18, $18, $18, $7e, $18, $18, $18,
$18, $18, $d8, $70, $00, $00
                byte $00, $18, $30, $60, $00, $78, $0c, $7c, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $0c, $18, $30, $00, $38, $18, $18, $18, $18,
$18, $3c, $00, $00, $00, $00
                byte $00, $18, $30, $60, $00, $7c, $c6, $c6, $c6, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $18, $30, $60, $00, $cc, $cc, $cc, $cc, $cc,
$cc, $76, $00, $00, $00, $00
                byte $00, $00, $76, $dc, $00, $dc, $66, $66, $66, $66,
$66, $66, $00, $00, $00, $00
                byte $76, $dc, $00, $c6, $e6, $f6, $fe, $de, $ce, $c6,
$c6, $c6, $00, $00, $00, $00
                byte $00, $3c, $6c, $6c, $3e, $00, $7e, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $38, $6c, $6c, $38, $00, $7c, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $30, $30, $00, $30, $30, $60, $c0, $c6,
$c6, $7c, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $fe, $c0, $c0, $c0,
$c0, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $fe, $06, $06, $06,
$06, $00, $00, $00, $00, $00
                byte $00, $c0, $c0, $c2, $c6, $cc, $18, $30, $60, $ce,
$9b, $06, $0c, $1f, $00, $00
                byte $00, $c0, $c0, $c2, $c6, $cc, $18, $30, $66, $ce,
$96, $3e, $06, $06, $00, $00
                byte $00, $00, $18, $18, $00, $18, $18, $18, $3c, $3c,
$3c, $18, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $36, $6c, $d8, $6c, $36,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $d8, $6c, $36, $6c, $d8,
$00, $00, $00, $00, $00, $00
                byte $11, $44, $11, $44, $11, $44, $11, $44, $11, $44,
$11, $44, $11, $44, $11, $44
                byte $55, $aa, $55, $aa, $55, $aa, $55, $aa, $55, $aa,
$55, $aa, $55, $aa, $55, $aa
                byte $dd, $77, $dd, $77, $dd, $77, $dd, $77, $dd, $77,
$dd, $77, $dd, $77, $dd, $77
                byte $18, $18, $18, $18, $18, $18, $18, $18, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $18, $18, $18, $18, $18, $18, $18, $f8, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $18, $18, $18, $18, $18, $f8, $18, $f8, $18, $18,
$18, $18, $18, $18, $18, $18
```

```
                byte $36, $36, $36, $36, $36, $36, $36, $f6, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $00, $00, $00, $00, $00, $00, $00, $fe, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $00, $00, $00, $00, $00, $f8, $18, $f8, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $36, $36, $36, $36, $36, $f6, $06, $f6, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $36, $36, $36, $36, $36, $36, $36, $36, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $00, $00, $00, $00, $00, $fe, $06, $f6, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $36, $36, $36, $36, $36, $f6, $06, $fe, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $36, $36, $36, $36, $36, $36, $36, $fe, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $18, $18, $18, $18, $18, $f8, $18, $f8, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $f8, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $18, $18, $18, $18, $18, $18, $18, $1f, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $18, $18, $18, $18, $18, $18, $18, $ff, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $ff, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $18, $18, $18, $18, $18, $18, $18, $1f, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $00, $00, $00, $00, $00, $00, $00, $ff, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $18, $18, $18, $18, $18, $18, $18, $ff, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $18, $18, $18, $18, $18, $1f, $18, $1f, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $36, $36, $36, $36, $36, $36, $36, $37, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $36, $36, $36, $36, $36, $37, $30, $3f, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $3f, $30, $37, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $36, $36, $36, $36, $36, $f7, $00, $ff, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $ff, $00, $f7, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $36, $36, $36, $36, $36, $37, $30, $37, $36, $36,
$36, $36, $36, $36, $36, $36
```

```
                byte $00, $00, $00, $00, $00, $ff, $00, $ff, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $36, $36, $36, $36, $36, $f7, $00, $f7, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $18, $18, $18, $18, $18, $ff, $00, $ff, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $36, $36, $36, $36, $36, $36, $36, $ff, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $ff, $00, $ff, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $00, $00, $00, $00, $00, $00, $00, $ff, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $36, $36, $36, $36, $36, $36, $36, $3f, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $18, $18, $18, $18, $18, $1f, $18, $1f, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $1f, $18, $1f, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $00, $00, $00, $00, $00, $00, $00, $3f, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $36, $36, $36, $36, $36, $36, $36, $ff, $36, $36,
$36, $36, $36, $36, $36, $36
                byte $18, $18, $18, $18, $18, $ff, $18, $ff, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $18, $18, $18, $18, $18, $18, $18, $f8, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $1f, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff,
$ff, $ff, $ff, $ff, $ff, $ff
                byte $00, $00, $00, $00, $00, $00, $00, $ff, $ff, $ff,
$ff, $ff, $ff, $ff, $ff, $ff
                byte $f0, $f0, $f0, $f0, $f0, $f0, $f0, $f0, $f0, $f0,
$f0, $f0, $f0, $f0, $f0, $f0
                byte $0f, $0f, $0f, $0f, $0f, $0f, $0f, $0f, $0f, $0f,
$0f, $0f, $0f, $0f, $0f, $0f
                byte $ff, $ff, $ff, $ff, $ff, $ff, $ff, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $76, $dc, $d8, $d8, $d8,
$dc, $76, $00, $00, $00, $00
                byte $00, $00, $78, $cc, $cc, $cc, $d8, $cc, $c6, $c6,
$c6, $cc, $00, $00, $00, $00
                byte $00, $00, $fe, $c6, $c6, $c0, $c0, $c0, $c0, $c0,
$c0, $c0, $00, $00, $00, $00
                byte $00, $00, $00, $00, $fe, $6c, $6c, $6c, $6c, $6c,
$6c, $6c, $00, $00, $00, $00
```

```
                byte $00, $00, $00, $fe, $c6, $60, $30, $18, $30, $60,
$c6, $fe, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $7e, $d8, $d8, $d8, $d8,
$d8, $70, $00, $00, $00, $00
                byte $00, $00, $00, $00, $66, $66, $66, $66, $66, $7c,
$60, $60, $c0, $00, $00, $00
                byte $00, $00, $00, $00, $76, $dc, $18, $18, $18, $18,
$18, $18, $00, $00, $00, $00
                byte $00, $00, $00, $7e, $18, $3c, $66, $66, $66, $3c,
$18, $7e, $00, $00, $00, $00
                byte $00, $00, $00, $38, $6c, $c6, $c6, $fe, $c6, $c6,
$6c, $38, $00, $00, $00, $00
                byte $00, $00, $38, $6c, $c6, $c6, $c6, $6c, $6c, $6c,
$6c, $ee, $00, $00, $00, $00
                byte $00, $00, $1e, $30, $18, $0c, $3e, $66, $66, $66,
$66, $3c, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $7e, $db, $db, $db, $7e,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $03, $06, $7e, $db, $db, $f3, $7e,
$60, $c0, $00, $00, $00, $00
                byte $00, $00, $1c, $30, $60, $60, $7c, $60, $60, $60,
$30, $1c, $00, $00, $00, $00
                byte $00, $00, $00, $7c, $c6, $c6, $c6, $c6, $c6, $c6,
$c6, $c6, $00, $00, $00, $00
                byte $00, $00, $00, $00, $fe, $00, $00, $fe, $00, $00,
$fe, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $18, $18, $7e, $18, $18, $00,
$00, $ff, $00, $00, $00, $00
                byte $00, $00, $00, $30, $18, $0c, $06, $0c, $18, $30,
$00, $7e, $00, $00, $00, $00
                byte $00, $00, $00, $0c, $18, $30, $60, $30, $18, $0c,
$00, $7e, $00, $00, $00, $00
                byte $00, $00, $0e, $1b, $1b, $18, $18, $18, $18, $18,
$18, $18, $18, $18, $18, $18
                byte $18, $18, $18, $18, $18, $18, $18, $18, $d8, $d8,
$d8, $70, $00, $00, $00, $00
                byte $00, $00, $00, $00, $18, $18, $00, $7e, $00, $18,
$18, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $76, $dc, $00, $76, $dc,
$00, $00, $00, $00, $00, $00
                byte $00, $38, $6c, $6c, $38, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $18, $18, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $00, $18, $00,
$00, $00, $00, $00, $00, $00
```

```
                byte $00, $0f, $0c, $0c, $0c, $0c, $0c, $ec, $6c, $6c,
$3c, $1c, $00, $00, $00, $00
                byte $00, $d8, $6c, $6c, $6c, $6c, $6c, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $70, $d8, $30, $60, $c8, $f8, $00, $00, $00,
$00, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $7c, $7c, $7c, $7c, $7c, $7c,
$7c, $00, $00, $00, $00, $00
                byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
$00, $00, $00, $00, $00, $00
```

Object Reference

These objects are organized in the same format as OBEX, but are not limited to those found in the Object Exchange.

Data Storage

- Basic I2C Driver
- Fat16 routines with secure digital card layer
- i2c object
- Memory Stick Datalogger

Display

- Graphics
- AiGeneric Video Driver (40x24 composite 16 color text)

Protocol

- PropTCP
    - Sockets Layer
    - HTTP Server
- Full Duplex Serial

Signal Generation

Fun

Tool

- FemtoBASIC

Human Input

- Atari Joystick and Virtual NES Driver
- RCTIME Object

Math

Sensor

- RCTIME Object

Speech & Sound

- (Hss) Hydra Sound System (7 channel audio/playback system)

One man Unix on the pPropQL and pPropQL020

Note: The original source of OMU can be found here. Thanks Steve !
Note: This is a work in progress, so the code may change without notice.

OMU was developed (it seems) on some kind of Codata board using a version of Unix with the tools and libc available at the time. Porting means using another compiler, another hardware, a new libc, executable format and filesystem.

A discussion about this project can be found at Parallax' boards here

Tasks

Getting a usable compiler

- Grab binutils (2.17 or 2.18) from gnu.org
- Grab gcc-4.1.2+ from gcc.gnu.org.

The core package is enough. Newer versions need and will compile some libraries that are certainly not needed.

To compile you will need the normal: make (gnu make), an installed compiler (gnu), autotools, flex, yacc or bison and so on.

Let's say that all happens in a prg directory:

$ cd prg

$ bzip2 -d
$ bzip2 -d
$ mkdir gcc-m68k

$ cd gcc-m68k

$ mkdir binutils

$ cd binutils

$ ../../bintutil-2.17/configure –target=m68k-unknown-elf

$ make [-j 3]

$ sudo make install

ready bintuils, if no errors. Fix the errors and redo. sometimes removing everything and relaunching configure helps if you installed some missing program/library.

$ cd ..

$ mkdir gcc

$ cd gcc

$ ../../gcc-4.1.2/configure –target=m68k-unknown-elf –disable-libssp

$ make [-j 3]

$ sudo make install

- Getting the source of OMU to at least compile with gcc:

Grab it here

Well it compiles... and gives loads of warnings :-(
Do not worry we will take care of them :-). Now, I ported all assembler code to use gcc syntax. Added some missing headers from linux (!) (they are also GPL v2) but to test I'd rather use a simulator, a love simulators :-). The mixed headers need to be sorted. Minimal header would be a better option: Linux' are just messy. Too many of them. Maybe an older 1.x version has better headers.

I'm now at the point where I have some 68k simulator that needs a usable front-end. I started to write a simulator in python from scratch, but I rather use something already done for speed reasons, of development.
Porting the code means now: getting the kernel to run to the point where it wants to execute init. For that some routines (libc related) have to be written. As well as a usable filesystem and a binary executable format has to be defined and incorporated. I think the best would be to use the elf format. It supports everything we need and want albeit it may contain too many features, but we can cross compile to it from either GNU/linux, *BSD, Solaris or Mac OSX.

Filesystem

pPropQL has a bootloader that has to load the kernel image from SD card (serial port could also be used). The filesystem recognition/read is built-in in this minimal bootloader. For QL a FAT fs could be used but for OMU a Un*x fs would be more appropriate. There are several options, all of them more or less complicated:

- minix
- ext2
- ufs (there are several of these, which one ?)

A possibility would be to have a filesystem contained in a file that resides in a FAT formatted SD card. A couple of programs could be written to manipulate this image. But that does not help with selecting a filesystem type. To keep it simple I'd choose minix. I think it represents the simplest of the bunch, especially considering that ufs is a name for several similar but different implementations. Minix sources for the filesystem are some 200 kbytes. A read-only mounter does not really need much more than the reading the superblock, the inode table and the root directory.

After some thoughts and digging of the sources, I decided that a minix either filesystem in an image or in a partition will be the way to go. A V1 filesystem can be as large as 64 MB, more than enough to host the entire system.

The geometry of the image is as follows:

boot block	1 block
superblock	1 block
bitmap	1 to 8 blocks, 1 bit per block, 65536 bits max, 8 kbytes max
zone map	same as bitmap ?
inode table	each inode occupies 32 bytes
free blocks	the rest of the blocks

The function of the zone in a V1 fs is not yet clear to me. I'll have to look closer into the mkfs utility. The filesystem is divided in blocks with the first block reserved for a boot block, I believe. Each block is 1 kbytes, V2 can have larger blocks and 32 bit pointers to blocks.

The filesystem described in the headers of OMU closely resembles this, so minimum changes should be needed.

Drivers

The most important part for propeller users is probably the interface between the propeller and a third

processor. the hardware interface, level shifting and so on has been described in the pPropQL and pPropQL020 pages. Basically the propellers act as memory mapped devices, with an address space, a data bus and read/write strobe signals that are asserted when the devices needs to be read or written.

On the propeller side and depending on the number of peripherals one or more COGs can listen to this strobe signals and act accordingly.

Let's see some examples.

ROM emulator

The propeller can act as a ROM. For this it only needs to serve a byte of data for every address. Using the multiplexed BUS employed in the mentioned boards, a simple routine that listens to the state of a read strobe can be used:

```
DAT
                        org     $0

ROMEMU                  mov     OUTA, #0
                        mov     DIRA, c0_c_DIRA

c0_romemu               waitpne c0_c_PROMCS, c0_c_PROMCS ' waits for C
S to be asserted
                        mov     c0_v_addr, INA          '@ 2   gets lo
w part of address
                        shr     c0_v_addr, #16          '@ 6
                        and     c0_v_addr, #255         '@10
                        add     c0_v_addr, PAR          '@14 adds ROM
offset
                        mov     c0_v_addrh, INA         '@18 now it is
 safe to get high addr
                        shr     c0_v_addrh, #8          '@22
                        and     c0_v_addrh, c0_c_MSKADDRH
                        add     c0_v_addr, c0_v_addrh   '@30
                        rdbyte  OUTA, c0_v_addr    '@34
                                                        '@56 (max)
                        or      DIRA, c0_c_DATAOUT      '@60
                        waitpeq c0_c_PROMCS, c0_c_PROMCS'@64
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_romemu
```

```
c0_c_DIRA               long    0
c0_c_MSKADDRH           long    $00007f00
c0_c_PROMCS             long    1<<25         ' NROMCS, ROM read strobe
, active low
c0_c_DATAOUT            long    $ff
c0_v_addr               long    0
c0_v_addrh              long    0
c0_v_data               long    0
```

The address is sent low byte first high byte next 2 M68K cycles apart. The program waits for the strobe signal to go high before disabling the output. **OUTA** can be used as destination register because no other output PIN is been used by this COG.

Video Memory

Using a propeller as a video generator is one of the most easy to implement functions. The propeller has special circuitry designed to generate video signals freeing the COGs for this time consuming task.

The following example shows how the propeller can act as memory mapped buffer (only write is shown). The buffer is limited to 4 kbytes because only text is implemented.

```
DAT
                        org     $0

VIDEOCOG                mov     DIRA, #0

c2_videoemu             waitpne c2_c_VIDEOW, c2_c_VIDEOW ' waits for N
VIDEOW to be asserted

                        mov     c2_v_addr, INA          '@ 2   gets lo
w part of address
                        shr     c2_v_addr, #16          '@ 6
                        and     c2_v_addr, #255         '@10
                        add     c2_v_addr, PAR          '@14   adds vi
deo buffer offset

                        mov     c2_v_addrh, INA         '@18   now it
is safe to get high addr
                        mov     c2_v_data, c2_v_addrh
                        shr     c2_v_addrh, #8          '@26
                        and     c2_v_addrh, c2_c_MSKADDR
                        add     c2_v_addr, c2_v_addrh   '@34
                        wrbyte  c2_v_data, c2_v_addr    '@38
```

```
                        waitpeq c2_c_VIDEOW, c2_c_VIDEOW
                        jmp     #c2_videoemu


c2_c_VIDEOW             long    1<<26       ' NVIDEOW strobe input, ac
tive low
c2_v_addr               long    0
c2_v_addrh              long    0
c2_v_data               long    0
c2_c_MSKADDR            long    $00000f00   ' only 4 kbytes !!!!
```

A extra COG generates the corresponding video signal and produces the image according to the data in this buffer (pointed by **PAR**). Graphic memory can also be used in the same manner, but probably a bigger buffer could be needed.

Other IO

Connection of keyboard, serial interface, RTC, SD/MMC card reader, Timers and so on can be accomplished in a similar manner to the one described above. One COG listens to reads and a second one to writes. As 2 or 3 HUB accesses need to be done a cycle time of about 1 microsecond is needed. This can be wasteful in some cases but adds simplicity in circuit design. More performance can be obtained using an AVR32, ARM or ColdFire processor instead.
One possible method is for every peripheral to listen to the read or write strobes directly. That can cause some problems and missed reads/writes. A better method is to dedicate 2 COGs to interface and the rest are free to interface with the mentioned devices.

A simple listener that interfaces with other COGs for the different tasks is shown below. Be aware of the fact that the normal FullDuplexSerial object has been modified to use two pointers instead of three (buffer and pointers inside the buffer). The keyboard object can be used as it is and the SD/MMC code is the newest mb_spi by Rokicki/Lonesock. Other peripherals have not yet been implemented. Buffer pointers are written during initialization. This method works because the peripherals have a shared memory interface.

```
DAT
                        org     $0

' This COG accepts reads from the processor

IORCOG                  mov     c0_p_rxbuffend, c0_p_rxbuff
                        add     c0_p_rxbuff, #RXBUFFLEN

c0_iorloop              waitpne c0_c_IOR, c0_c_IOR      ' waits for IO
```

```
R to be asserted
                        mov     c0_v_addr, INA          ' @ 2 cycles a
fter IOR is asserted
                        shr     c0_v_addr, #16-4        ' @ 6
                        and     c0_v_addr, #$1f0  wz    '@ 10
            if_nz       jmp     c0_v_addr

                        mov     OUTA, c0_v_ready        ' command 0 re
ads status of engine A5 == ready
                        or      DIRA, c0_c_DATAOUT      ' activates ou
tputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop

c0_v_ready              long    0

                        long    0[(((($+15)>>4)<<4)-$]

{ This must be at address 01
  Receive status read
}
                        ' Port 1
c0_serrx_01             rdlong  c0_p_ptr2, c0_p_rxtail
                        rdbyte  OUTA, c0_p_ptr2             ' reads la
st byte received
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        cmp     c0_p_ptr2, c0_p_rxbuffend   wz
            if_z        mov     c0_p_ptr2, c0_p_rxbuff      ' moves ta
il to beginning of buffer
            if_nz       add     c0_p_ptr2, #1
                        wrlong  c0_p_ptr2, c0_p_rxtail      ' saves ne
w tail of buffer
                        jmp     #c0_iorloop

c0_p_rxhead             long    0
c0_p_rxtail             long    0
c0_p_rxbuff             long    0
c0_p_rxbuffend          long    0

                        long    0[(((($+15)>>4)<<4)-$]

{ returns $0f if there are no bytes to read
```

```
}

c0_serstatus_02         rdlong  c0_p_ptr2, c0_p_rxtail
                        rdlong  c0_p_ptr1, c0_p_rxhead
                        cmp     c0_p_ptr1, c0_p_ptr2   wz
                        muxz    OUTA, #$0f
                        or      DIRA, c0_c_DATAOUT
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop

c0_p_kbdhead            long    0
c0_p_kbdtail            long    0
c0_p_kbdbuff            long    0

                        long    0[(($+15)>>4)<<4)-$]

c0_kbdrx_03             rdlong  c0_p_ptr2, c0_p_kbdtail
                        rdlong  c0_p_ptr1, c0_p_kbdhead

                        add     c0_p_ptr1, c0_p_kbdbuff
                        rdbyte  OUTA, c0_p_ptr1
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        sub     c0_p_ptr1, c0_p_kbdbuff
                        add     c0_p_ptr1, #1
                        and     c0_p_ptr1, #15
                        wrlong  c0_p_ptr1, c0_p_kbdtail
                        jmp     #c0_iorloop

' Returns 00 if there are chars to read
' Returns 0F if there are no chars to read

                        long    0[(($+15)>>4)<<4)-$]

c0_kbdstatus_04         rdlong  c0_p_ptr2, c0_p_kbdtail
                        rdlong  c0_p_ptr1, c0_p_kbdhead
                        cmp     c0_p_ptr2, c0_p_ptr1   wz   ' are they
 ==
                        muxz    OUTA, #$0f
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
```

```
                        jmp     #c0_iorloop

c0_p_sdcmd              long    0
c0_p_sdblk              long    0
c0_p_sdptr              long    0
' data pointer
c0_v_dptr               long    0
c0_v_cnt                long    0
c0_c_512                long    512

                        long    0[((($+15)>>4)<<4)-$]

                        ' dummy
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop

                        long    0[((($+15)>>4)<<4)-$]
' reads the error
' resets the data pointer to the beginning of the buffer
'
c0_readcmd_06           rdlong  c0_v_data, c0_p_sdcmd       ' reads er
ror
                        mov     c0_v_dptr, c0_p_sdptr       ' resets p
ointer
                        mov     c0_v_cnt, #0                ' counter
                        shr     c0_v_data, #24
                        mov     OUTA, c0_v_data             ' presents
 MSByte
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop

                        long    0[((($+15)>>4)<<4)-$]
' Reads the sector data
' after 512 reads the pointer is reset

c0_readdta_07           rdbyte  OUTA, c0_v_dptr             ' reads er
ror
                        add     c0_v_cnt, #1
                        cmp     c0_v_cnt, c0_c_512   wz
            if_z        mov     c0_v_cnt, #0
            if_z        mov     c0_v_dptr, c0_p_sdptr       ' resets p
ointer
```

```
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop

                        long    0[(($+15)>>4)<<4)-$]

' Allows to read the last written block number
c0_readblk_08           rdlong  c0_v_data, c0_p_sdblk       ' reads bl
ock number
                        shr     c0_v_data, #24
                        mov     OUTA, c0_v_data             ' presents
 MSByte
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop

                        long    0[(($+15)>>4)<<4)-$]

c0_readblk_09           rdlong  c0_v_data, c0_p_sdblk       ' reads bl
ock number
                        shr     c0_v_data, #16
                        mov     OUTA, c0_v_data             ' presents
 MSByte
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop

                        long    0[(($+15)>>4)<<4)-$]

c0_readblk_0A           rdlong  c0_v_data, c0_p_sdblk       ' reads bl
ock number
                        shr     c0_v_data, #8
                        mov     OUTA, c0_v_data             ' presents
 MSByte
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop
```

```
                        long    0[(($+15)>>4)<<4)-$]        ' == .alig
n 16

c0_readblk_0B           rdlong  c0_v_data, c0_p_sdblk       ' reads bl
ock number
                        mov     OUTA, c0_v_data             ' presents
 MSByte
                        or      DIRA, c0_c_DATAOUT          ' activate
s outputs
                        waitpeq c0_c_IOR, c0_c_IOR
                        andn    DIRA, c0_c_DATAOUT
                        jmp     #c0_iorloop




c0_c_DIRA               long    0
c0_c_MSKADDRH           long    $00ff0000
c0_c_IOR                long    1<<25
c0_c_DATAOUT            long    $ff

c0_v_addr               long    0
c0_v_addrh              long    0
c0_v_data               long    0
c0_p_ptr1               long    0
c0_p_ptr2               long    0

                        jmp     #c0_iorloop
                        fit     $1f0
```

More to come!

Important Note

When comparing the information here with that in the Propeller Chip Manual and Datasheet you will notice discrepancies. For example, this page states the PLL output is rated at 64MHz to 160MHz while the Manual states it is rated at 64MHz to only 128MHz, this page says an external crystal with PLL enabled can have a frequency of between 4MHz and 10MHz, the Manual states 4MHz and 8MHz.

The reason is that Parallax have conservatively rated the Propeller Chip to specify the circumstances in which it is, for want of a better phrase, "guaranteed to work". The reality is that Parallax themselves produce product which requires operation outside the conservative ratings given; the SpinStamp uses a 10MHz crystal with the PLL output running at 160MHz. By all accounts this has never proven to problematic.

In consequence, this page has been written with respect to "best known reality" rather than simply reiterating the specified ratings given in the Propeller Chip Manual or datasheet.

System Clock Speed

The internal operation of the Propeller Chip is controlled by a system clock which determines how quickly the Propeller runs and how fast the execution of programs are.

There are four sources from which a system clock can be derived -

- Internal slow oscillator (RCSLOW, 20kHz)
- Internal fast oscillator (RCFAST, 12MHz)
- Crystal or Resonator
- Oscillator Module

Additionally, when using a crystal, resonator or oscillator module a PLL may be enabled to increase the system clock speed by up to a factor of 16.

The system clock speed is rated at up to 80MHz and operation at up to 100MHz can usually be achieved under normal circumstances. As the clock speed increases above 100MHz the internal circuitry of the Propeller Chip will start to become unstable and correct operation is not guaranteed.

There is no definitive statement possible as to what the absolute maximum system clock speed is because that depends upon what methods are used to attain a particular system clock speed. A super-cooled Propeller Chip will likely be able to operate at much higher clock speed than one which is simply placed upon a PCB with no special considerations taken.
This also depends on the supply voltage. According to experiences 100MHz systems run quite reliable @ 3.6V but not well @ 3V.

RCSLOW

While notionally running at 20kHz, the manufacturing tolerances of the the RCSLOW oscillator means that, for any particular Propeller Chip, the oscillator may run at anywhere between 13kHz and 33kHz.

The oscillator should remain fairly stable over short periods of time but it may drift over longer periods and the actual oscillator frequency will depend on operating temperature and supply voltage.

While allowing operation with no additional, external components the RCSLOW oscillator will unlikely be usable in applications which require accurate timing without calibration and may require re-calibration during application execution. For accurate timing it is recommended to use an external crystal, resonator or oscillator module.

RCFAST

While notionally running at 12MHz, the manufacturing tolerances of the the RCFAST oscillator means that, for any particular Propeller Chip, the oscillator may run at anywhere between 8MHz and 20MHz.

The oscillator should remain fairly stable over short periods of time but it may drift over longer periods and the actual oscillator frequency will depend on operating temperature and supply voltage.

While allowing reasonable fast operation with no additional, external components the RCFAST oscillator will unlikely be usable in applications which require accurate timing without calibration and may require re-calibration during application execution. For accurate timing it is recommended to use an external crystal, resonator or oscillator module.

The Propeller Chip uses the RCFAST oscillator while it is booting and downloading programs from a PC. To cater for the variations between Propeller Chips the download protocol uses a mechanism which does not require absolute timing and is suitable for all Propeller Chips regardless of the operating frequency of the RCFAST oscillator.

Crystal or Resonator

The Propeller Chip supports the direct connection of a crystal or resonator with frequencies between DC and 80MHz.

When the PLL is enabled the crystal or resonator frequency must be between 4MHz and 10MHz.

Note that no capacitors are needed in the oscillator circuit and the oscillator can simply be connected to the appropriate Propeller Chip pins.

Oscillator Module

The propeller Chip supports the direct connection of an oscillator module with an operating frequency of between DC and 128MHz.

When the PLL is enabled the oscillator module operating frequency must be between 4MHz and 10MHz.

PLL

An internal PLL can be enabled when using a Crystal controlled oscillator or an oscillator module. When enabled it can boost the internal system clock speed by a factor of up to 16.

The PLL is a fixed "times 16" multiplier which is followed by a divider which can reduce the PLL output to create the system clock. The PLL output is rated from 64MHz up to 160MHz which requires that the crystal or oscillator module it is driven from is not below 4MHz and does not exceed 10MHz. The system clock is rated at up to 80MHz which requires that the PLL output is divided down to a value no greater than 80MHz.

Taking both these factors into account means that only certain crystal and oscillator module frequencies are allowed when the PLL is enabled. As already stated, the crystal or oscillator module frequency must not exceed 10MHz to keep the PLL output within rated specification. To run with a system clock speed of 80MHz, a 5MHz oscillator source, multiplied by 16 by the PLL and divided by one would be acceptable as would a 10MHz crystal, multiplied by 16 by the PLL and divided by two.

In practice the system clock speed is usually capable of reaching 96MHz under normal circumstances allowing a 6MHz oscillator source to be used with the PLL and a post-PLL division of one. Under most normal circumstances a system clock speed of 100MHz is attainable allowing the oscillator source to be 6.25MHz while still retaining a post-PLL division of one.

Parallax report that all Propeller Chips must pass soak testing at 104MHz system clock speed before being allowed out the factory door.

Some success has been reported using 7.3728MHz crystals on an otherwise unmodified ProtoBoard. Other reports are that there can be problems at this frequency when the Propeller is being worked hard.

Similar success has been reported using 14.31818MHz crystals with a PLL8x setting.

Using 15MHz is reported to present difficulties which would fit with hitting the 120MHz system clock speed limit where Parallax report break down in reliable Cog operation. The hard limit will almost certainly be affected by temperature, voltage and particular chip.

While it may be possible to achieve even higher system clock speeds under the right conditions (running above normal operating voltage, forced cooling or even freezing), operation under normal circumstances is not guaranteed above 80MHz.

Crystal Frequency	PLL divided by 2	PLL divided by 1
4MHz	32MHz	64MHz
5MHz	40MHz	80MHz
6MHz	48MHz	96MHz
6.25MHz	50MHz	100MHz
6.5MHz	52MHz	104MHz
7.3728MHz	59MHz	118MHz
8MHz	64MHz	Not allowed
10MHz	80MHz	Not allowed
14.31818MHz	114.5MHz	Not allowed

uOLED-96-PROP

There has been some debate and confusion over the capabilities of the uOLED-96-PROP manufactured by 4D Systems, a small graphics display module which includes an embedded Propeller Chip as its controller.

uOLED-96-PROP Mk 2 - 10MHz Crystal

The uOLED-96-PROP Mk 2 uses an 10MHz and has no reported issues. This can operate with up to PLL8x giving a maximum system clock of 80MHz.

uOLED-96-PROP Mk 1 - 8MHz Crystal

The uOLED-96-PROP Mk 1 uses an 8MHz crystal and manufacturer provided example code shows configuration using PLL with no post-PLL division; this results in a 128MHz system clock, well above what is "normally allowed".

The reasoning as to why this works is that the choice of Propeller Chip Packaging and board design mitigates the potential problems of high system clock speed.

In practice it has been demonstrated that some code appears to execute as expected at 128MHz but it has also been shown that some code will not and will fail in a manner as predicted by Parallax at such high system clock speeds. There is no clear-cut, "it does" or "does not" work at 128MHz, answer. It is a case of "Your mileage may vary" (YMMV) and will depend very much on what the actual code is.

For those who have a uOLED-96-PROP Mk 1 and are concerned with using a 128MHz system clock or are having problems when trying to use that, the simple solution is to configure the application code to use a PLL8x setting rather than the PLL16x setting. This will reduce the system clock speed to 64MHz but should be satisfactory for many application programs.

For those who wish to use the uOLED-96-PROP Mk 1 with a system clock speed above 64MHz, one option would be to replace the 8MHz crystal with another. This must be done with due care and will likely void any manufacturers warranty. It is not recommended that the crystal be changed unless you know what you are doing and fully understand the potential consequences; you are entirely responsible for any adverse effects or damage which may occur during such a process.

DLP-PROP

The DLP-PROP is an Propeller-based board manufactured by DLP Design which is provided as a DIP-style plug-in module, complete with USB programming interface which also provides power and acts as an oscillator module for the Propeller Chip.

There are two issues with this configuration which DLP-PROP users should be aware of -

- Firstly that the 6MHz oscillator clock provided by the USB interface is only present when the USB interface is connected and working properly (has been "enumerated" by the host PC), and,
- The 6MHz clock source is not as stable nor as accurate as a crystal controlled oscillator source.

The consequences are that the Propeller Chip will not function when not connected to a USB host system and accurate timing may not be possible.

Standard Configurations

The following table shows the crystals fitted and maximum system clock speeds attainable for a variety of commercial and third-party Propeller Chip boards ...

Board	Crystal	Max PLL	Max System Clock	Notes
Parallax DemoBoard	5MHz	PLL16x	80MHz	
Parallax PDB	5MHz	PLL16x	80MHz	1

Parallax ProtoBoard	5MHz	PLL16x	80MHz	
Parallax SpinStamp	10MHz	PLL8x	80MHz	
DLP-PROP	6MHz	PLL16x	96MHz	
Hydra Games System	10MHz	PLL8x	80MHz	
PropRPM	5MHz	PLL16x	80MHz	
PropSTICK	5MHz	PLL16x	80MHz	
uOLED-96-PROP Mk 1	8MHz	PLL8x	64MHz	2
uOLED-96-PROP Mk 2	10MHz	PLL8x	80MHz	

(1) Parallax Professional Development Board (PPDB)
(2) May be usable to 128MHz under some circumstances (see earlier notes).

The majority of boards use an 80MHz system clock speed (5MHz with PLL16x or 10MHz with PLL8x) and software examples are usually configured for 80MHz operation. It is often necessary to adjust the configuration settings for use with other boards.

Over-Clocking

The following tables show the performance gains which can be achieved by replacing a fitted crystal with another to increase the execution speed of the Propeller Chip ...

	To 80MHz	To 96MHz	To 100MHz	To 104MHz	To 118MHz
From 80MHz		+20%	+25%	+30%	+48%
From 96MHz	-16%		+4%	+8%	+23%
From 100MHz	-20%	-4%		+4%	+18%
From 104MHz	-23%	-8%	-4%		+14%

Packaging Propeller Software

Generally speaking, it is a good idea to publish complete packages for others to learn from. This page is about sharing tips and tricks for doing this. If you are into the prop (and that's a safe bet if you are here reading now), adding this value will grow the community of Propeller enthusiasts. The more the merrier**.**

Identify target Propeller Setup(s) and Clock speed(s)

With all the boards and home-brew setups out there now, it's beginning to make sense for this to occur. Demo board & Hydra both run @ 80Mhz. Hybrid and some others run @ 96Mhz. If your program is easily configured for multiple boards, that's great and can easily be put in the self-documenting code options. If this is not the case, then others will know what setup to use, or that they might need / want to port it to their setup.

Other things to include are the video output standard(s), such as NTSC, PAL, both!, VGA, etc...

eg: some_game_HYBRID_96_PAL.zip

Propeller IDE Tool Archive function

The propeller tool has an archive function that will grab all the objects needed for a particular project and zip them up into one simple package for others to try out, without having to locate dependent packages. Highlight your main program, then choose FILE, ARCHIVE and go from there. Archive includes a name and time stamp for easy tracking.

You must have successfully compiled your code, in order for the archive function to work. Just load your main program, and compile, then archive.

Additional documents, in the form of spreadsheets, text documents, screen captures, sample assets, etc... can be bundled with the archive created with the Parallax Propeller IDE tool, using your archive manipulator of choice.

Combine everything into one SPIN wrapper

Bundling things together as a demo that can also be used as a building block object is another ready

option. This may require some greater effort, on the part of others, to utilize the work. YMMV.

Self-documenting code options

The Propeller tool includes a nice font that can render schematics, flow charts and general ASCII text comments. Consider verbose comments in your code, particularly if you elect to produce one combined file. Users can then run the demo, read the comments, tweak, then apply the bits that make sense.

Use the SPIN block comment to embed HTML documentation right in the program file. Good for simple graphical elements, color palettes, etc...

Hydra CD

Really great games and game related utilities, well packaged and documented, is likely to end up on the HYDRA CD. Quick and easy fame can be yours!

Some objects are distributable, some are not.

The sample code on the HYDRA CD, written by Andre's demo team is not freely distributable. Note these files in your code documentation, by name so users can locate them after having obtained their own copy.

Hanno has given permission to include the 'Conduit' object to provide ViewPort support in your packages. This means that all of the ViewPort code used to test the spin code can stay in the distributed version, even if the end user does not have ViewPort. Since the name 'Conduit' is the same across all versions of ViewPort, but the software interface that it supports does not, be careful not to have users place the copy of conduit that you provide anywhere that it might over-write a previously installed version. if the user already has ViewPort and Conduit installed, then they should just delete the version that you have packaged up.

Other files, published in the Parallax Object Exchange are distributable, as is the Parallax reference material. Web forum code postings vary considerably on this, making it best to just ask before distributing these. If your project modifies something, it is a really great idea to modify the name so people do not confuse objects. It is also a good idea to reference the original in the program header as well, so people can follow the logic later.

Consider staking out a name space prefix for your code, and use version numbers or the archive timestamps for people to track changes.
eg: potatohead--wonderdriver_001.spin

Consider not removing incremental code changes from the Wiki or forums. These are often very helpful to others for learning how things evolved in addition to just learning how to use the code in it's current release form, The HYDRA book and CD illustrate this idea perfectly, with incremental changes to drivers both in the text and on CD. You may also find your own pre-release efforts useful at some later time. If nothing else, it makes a great off-site backup lest disaster strikes!

Include running screen shots

It's helpful to see what the expected result is! A coupla JPEGs from your capture card, camera, or screen capture code running on the Propeller itself, help the potential user to see what it is they will be running. Colors, etc... can be compared to, just in case setups differ somehow.

Specify your license

By default, everything is copyrighted. Take a moment to include your intent in the program header, along with version, purpose, name, date, etc... This simple bit of meta data makes things easier for everyone. Hobby level developers are not often concerned with these matters, but a great many others are.

The share and share alike culture surrounding the Propeller is a big value add. While not always possible, where it is, an explicit statement can save someone some grief and maybe see your code used, improved on, or just found useful by others.

As the software creator, you are entitled to specify most any conditions of use. Code lacking such specifications may not be useful to others having to work in a more formal environment. This can be a simple note, to a fairly complete specification, such as BSD, GPL, Creative Commons, etc... If you can, keep this really simple. Others looking at the code, for the first time, can grok right away, what their options are, and can know to ask for greater permission, if their intent warrants that.

Choosing a License

The complexities of licensing and implications of particular licenses are subjects which cannot be fully covered here, but authors of software should bear in mind that a chosen license does not just grant permission for things which can be done, but also places restrictions on what cannot and may also have consequential impact on anyone using your code which they, or you, may find undesirable.

Many software authors place their work in the public domain so others may use that code and benefit from it, however, a poorly chosen license may restrict its use more than intended. Some licenses may require anyone using your code to also publish the work in which they use your code. This may be

unsatisfactory for some who may wish to use your code but will be unable to do so. You may wish to impose such obligations or you may not (it is a matter for the author to decide) but the author should be aware of the consequences of whatever license they choose to use and ensure it is suitable for and matches what they intend.

Before choosing a license, it is recommended that software authors decide what they wish to allow and what they wish to prevent and researches how any particular licenses match with their desires. Do not choose a license simply because it is the 'flavor of the month' or others say it is the preferred license to use; choose a license which matches your own intent.

NTSC Palette Mode

The propeller's build in video hardware is designed for displaying graphics in 2 color mode or 4 color mode NTSC.

2 Color Mode

Using this mode a program can display 2 colors per 32 horizontal pixels.
Data is sent to the video hardware using a **WAITVID palette,pixels** instruction. **palette** is a long value with a palette of 2 colors.

Byte 3 (MSB)	Byte 2	Byte 1	Byte 0 (LSB)
don't care	don't care	Color 1	Color 0

data is a long containing 32 single bit pixels. The color displayed is looked up from the palette. The least significant bit is displayed first.

4 Color Mode

Using this mode a program can display 4 colors per 16 horizontal pixels.
Data is sent to the video hardware using a **WAITVID palette,pixels** instruction. **palette** is a long value with a palette of 4 colors.

Byte 3 (MSB)	Byte 2	Byte 1	Byte 0 (LSB)
Color 3	Color 2	Color 1	Color 0

pixels is a long containing 16 two-bit pixels. The color displayed is looked up from the palette. The least significant bit-pair is displayed first.

Hi-Color Mode

This is a trick used for a mode that can display all 86 colors on any pixel. The VCFG register is set up for a 4 color mode, but the VSCL register is set up such that it only outputs 4 pixels per WAITVID. Data is then sent to the video hardware using a **WAITVID palette,#%%3210** instruction. So each of the 4 pixels will always display one of the 4 palette colors. This means that we can just set the palette to the 4 colors we want to display in order.

e.g. To display 4 pixels of black ($02), white ($06), yellow ($5D) and blue ($FB):
WAITVID colors,#%%3210

```
' %%3210 == 11 10 01 00 (4 pixels pattern)
' 1st pixel will be color 0 ($02, black)
' 2nd pixel will be color 1 ($06, white)
' 3rd pixel will be color 2 ($5D, yellow)
' 4th pixel will be color 3 ($FB, blue)
...
colors LONG $02_06_5D_FB
```

Propeller Assembler Source-code Debugger

The Propeller Assembler Source-code Debugger (PASD) is a suite of software components which enable end-users to debug Propeller assembly language code at the source level using a remote (USB attached) Windows PC.

Author

Andy Schenk (Ariba)
Manual by Eric Moyer (epmoyer)

License

Website

Download from Insonix
Introduced on the Parallax forum here.

Overview

(from the manual)
PASD supports setting multiple break points, single-step execution, memory inspection/modification of COG RAM, inspection of Main RAM, label recognition, and I/O pin state inspection. The debugger suite consists of a Windows application, a spin object and a short Debug Kernel which must be inserted at the beginning of the code to be debugged. The Debug Kernel is only 12 longs in size, and makes possible communication with the PASD spin driver, which runs into own Cog. The PASD spin driver communicates over the Propeller's serial programming interface with the PASD Windows application running on an attached PC. Except for pins 30 and 31 (the Propeller's serial programming interface pins) all Propeller IO pins are freely available during debugging.

The total Propeller resource footprint of the PASD suite is:
1) Two IO Pins (30 and 31, the serial programming interface pins).
2) 12 longs at the start of COG Ram in the COG whose assembly code is being debugged.
3) The upper two longs of Main RAM ($7ff8 and $7fff).
4) The PASD driver which occupies about 223 Longs and runs in one dedicated COG.
All remaining Propeller resources (cogs, RAM, pins) are fully usable.
PASD presently supports debugging code in only one COG at a time.

This page is under construction. It is intended to contain information for people designing printed circuit boards for the Propeller.

Currently a lot of placeholder text and links to forum threads. Will be summarizing info from threads as time goes by.

This page is primarily maintained by mpark, but anyone is welcome to make improvements.

Propeller chips

- Connect all Vdd and Vss. Floating Vdd or Vss pins may cause PLL failure [citation needed].
- "What destroys the PLL (actually, the logic circuits downstream from the PLL) is high current running between VDD or VSS pins. Those pins need to be tied together very closely on the PCB. As Leon said, good PCB layout is critical. Bad PCB layout can not only cause caps to be ineffective, but give current the opportunity to flow between VDD or VSS pins, which can damage the Propeller. The key is to keep things very tight. Make power routing your first priority in a PCB design. Make power traces as short as physically possible, and at least 15 or 20 mils wide."—Chip Gracey http://forums.parallaxinc.com/forums/default.aspx?f=25&m=410277
- Best practice Power/Ground on Propeller Chip
- Connection of RESn Pin
- Pins 30/31

Crystal

- The Propeller specs call for a parallel-resonant crystal with about a 20pF load capacitance (for 5MHz). If you were to substitute a series-resonant crystal or one with the wrong load capacitance, you might well see a frequency discrepancy.*
- crystal considerations thread
- 5MHz quartz with small housing required (Note: If using small watch crystal, clean out any flux or other gunk between the closely-spaced pads. Cautionary tale: What's wrong with my PCB? [re-resolved])

EEPROM

- The Prop has to have an EEPROM with at least 32K bytes of storage like the AT24C256 or Microchip 24LC256. If you need to store any significant amount of data or another program (overlay), you'd need an AT24C512 or Microchip 24LC512 or possibly an AT24C1024B or Microchip 24LC1025.
- Alternative: replace the eeprom with a ramtron FM31L278 device; you will have the same boot-eeprom capability (32K-unlimited writes) plus a real-time clock.*

Bypass/decoupling caps

- You need one 0.1uF ceramic bypass cap next to each Vdd/Vss pair.
- Design rules for stable propeller operation and overclocking
- decoupling caps on the prop. If you plan to socket the Propeller, you can fit them under the propeller, that lets you get them really close to the power supply pins. I'd do one for each side. 0.1 uf or so. I've found that Machine pin 40 pin dip sockets give you the room, the cheap sockets put a reinforcement rib right across the center, blocking the area I normally put them. My point? look at the socket you plan to use before laying out the board, or you may find problems when you decide to solder them together.
- about decoupling caps ... put down some 1208 or 806 smd device pads with short fat traces. There is nothing on the back so a ground plane would be nice there and will make the fab easier to spin. If it was my board I would allow a provision for putting smd resistors in series with all propeller traces.
- Decoupling caps, the magic component

Power

- Power connector: 2.1mm center positive is what most users will have on hand.
- Power Supply Design for Propeller
- (about reverse polarity protection)
- "CP-202A-ND has bent pins and CP-102A-ND has smaller pins (in the pcb). Both are 2.1mm and from Digikey"*

Power connector @rapidonline, @digikey

General

- Avoid 90 degree bends in traces.
- The main reason for avoiding 90 degree track angles is that it makes etching harder - you are more likely to get shorts, especially with narrow tracks and spacing. Mitered tracks look neater, as well.
- Other reasons not to use 90 angles is traces get narrowed at corners and they increase reflections in high frequency traces.*
- I would recomend if people have room have an sd card and 512kbit eeprom for comercial products. if you do then you can use my bootloader to install updates. *
- A good rule of thumb is to keep the traces no closer than 0.050 in from the edge of the board.
- Ultimate Guide to Inkjet Direct PCB Printing
- making your own printed circuit boards?
- Making PCB's
- http://www.instructables.com/id/Professional-PCBs-almost-cheaper-than-making-them-/

Surface mount

- stencils from http://ohararp.com/
- reflow ovens are expensive so i use a convection toaster oven as one. works just as good but

requires some babysitting.
- 0603 resistors are very easy to work with by hand and their smaller size makes for easier placement. *
- dissenting opinion: "0603 parts are tiny, I think that unless space is a constraint 0805 or 1206 are ok. Just making them smaller may not always be advantageous." *
- Soldering 0805 SMD resistors and QFP-52 ICs question
- SMB board design (mostly about solder)
- Solder Paste -Where can I get some?
- Propeller Controlled Reflow Oven
- (discussion of stencils)
- thread contains "Prop_44 pin custom component for your ExpressPCB library"

PCB fabrication houses

Advanced Circuits	in Colorado*
BatchPCB	"is the cheapest place but it does take some time to get the boards back (they are actually made in China)"; "Nuts and volts uses a service called batch pcb I think. They take small orders until they have enough for a panel and then run it. The turnaround time varies but the price is reasonable."*; "I don't think it is Nuts and Volts that's directly affiliated with Batch PCB, I think it is www.sparkfun.com. If you have a lot of boards that add up to one or more panels, then Batch PCB can turn around fairly fast. I believe Batch PCB uses Gold Phoenex for the actual fab." *
ExpressPCB	
http://www.futurlec.com/PCBService.shtml	"They have an automated quote generator on-line. Reasonable prices, fast turn-around, low cost World wide shipping available.*
Gold Phoenix	in China; "If you need large panels"*
PCBexpress	
SeeedStudio	in China; "has a nice PCB service and can even partner with the designer to distribute an open hardware "product" through their online store. Low cost International shipping with tracking via Hong Kong." *

Related thread: Best Method for "short run" of PCBs

PCB design software

- Eagle
- Diptrace
- Express PCB
- FreePCB
- PCB123
- Advanced Circuits
- PCB Software and Production

You forgot to mention the best (IMHO), free (GPL), cross platform (MAC too) suite of programs for EDA (Electronic Design Automation), KiCAD. KiCAD has Schematic capture, PCB layout editor, Gerber viewer, footprint selector, footprint editor. autorouter, project manager, etc.; a complete suite capable of producing professional quality layouts..

iut-tice.ujf-grenoble.fr/kicad/

kicad.sourceforge.net/wiki/index.php/Main_Page

There's a large (2,5k member) active KiCad Yahoo Group here:

tech.groups.yahoo.com/group/kicad-users/

Parallax approved Eagle parts

Propeller Schematic Symbols for Express PCB

There is an excellent Homebrew PCB Yahoo Group with almost 5k members. Much help on EDA and fabricating your own PCBs:

tech.groups.yahoo.com/group/Homebrew_PCBs/

Photoboards: They really need testing. Exposure time varies from manufacturer to manufacturer (and with the kind of paper or transparency you have), and developing too.*

Common numbers

- Reminder: Units: Parts come in imperial and metric sizes, sadly. When you give size it would be useful to have both for example: 50 mils (1.27mm), or 1.27 mm (0.05"). Or something similar.
- standard via size?
- I'd make your prop plug via's (holes) 1.02 mm or so - the standard size is just too small for a standard header.
- demo board pinouts

- protoboard dimensions
- DIP pin spacing

Members' boards

- LucidGuppy's Eagle Schematic Reference Design
- Cluso99's TriBladeProp
- Phildapill's PropBoard thread
- WBA Consulting's uSD datalogger
- Cenlasoft's
- eagletalontim: wanting to get started
- Microcontrolled's mobile device development kit ideas
- Sal Ammoniac's board

Other threads of interest

- PCB drill bits
- FTDI reset bug
- buck/boost converters
- I/O cycle timings
- Minimal TV or VGA pins
- 3.3V step-up IC for Propeller apps?
- The proper way to clean PCBs...before & after soldering
- Advice on hot air rework station
- Double sided but not... [Now a review of my new PCB Fab in a Box kit! pg 2
- Looking for help with PCB assembly

Rayman's PCB page

This is the PinDefs.spin standard. *Proposed PinDefs.spin 1.0 Standard.*
At this point it is uncommitted, and all are invited to make suggestions to how byte configuration should be identified.

```
CON
   '' PinDefs.spin configuration file, adjust or overwrite as needed.

   _clkmode  = xtal1 + pll16x
   _xinfreq  = 5_000_000

   'NTSC/PAL Switch
   PAL_MODE = 0 | 50 | 60   '0=NTSC, 50/60 = PAL with 50 or 60 Hz.
   ' Primary Keyboard Single
   KEYBOARD1    = 26
   ' Primary Mouse Connection Single
   MOUSE1       = 24
   ' Primary TV Connection Single
   TV_DAC1      = 12
   ' Primary VGA Connection Single
   VGA1         = -1
   ' Primary Audio Connection Single
   AUDIO1       = 10
   ' Secondary Audio Connection
   AUDIO2       = 11

   ' Primary SD media Connections
   FSRW1_DO     = 16
   FSRW1_Clk    = 17
   FSRW1_DI     = 18
   FSRW1_CS     = 19

   'DS1302 Clock Settings
   DS1302_INCLK  = 0
   DS1302_INIO   = 0
   DS1302_CS     = 0

   ' Serial I/O
   SERIAL1_TX    = 0
   SERIAL1_RX    = 0

   ' XBee Configuration
   XBEE1_TX      = 0
   XBEE1_RX      = 0
```

```
' HYDRA NET Configuration
HYDRANET_TX   = 0
HYDRANET_RX   = 0

' IR Emit/Detect
IRDETECT      = -1
IREMIT        = -1

' Ping Sensor
PING_DATA    = -1

'GPS Sensor
GPS_DATA     = -1

' NES bit encodings general for state bits
NES_RIGHT     = %00000001
NES_LEFT      = %00000010
NES_DOWN      = %00000100
NES_UP        = %00001000
NES_START     = %00010000
NES_SELECT    = %00100000
NES_B         = %01000000
NES_A         = %10000000

' NES bit encodings for NES gamepad 0
NES0_RIGHT    = %00000000_00000001
NES0_LEFT     = %00000000_00000010
NES0_DOWN     = %00000000_00000100
NES0_UP       = %00000000_00001000
NES0_START    = %00000000_00010000
NES0_SELECT   = %00000000_00100000
NES0_B        = %00000000_01000000
NES0_A        = %00000000_10000000

' NES bit encodings for NES gamepad 1
NES1_RIGHT    = %00000001_00000000
NES1_LEFT     = %00000010_00000000
NES1_DOWN     = %00000100_00000000
NES1_UP       = %00001000_00000000
NES1_START    = %00010000_00000000
NES1_SELECT   = %00100000_00000000
NES1_B        = %01000000_00000000
NES1_A        = %10000000_00000000

' Nes Controller I/O Configuration
JOY_CLK       = 3
```

```
    JOY_SHLDn     = 4
    JOY_DATAOUT0  = 5
    JOY_DATAOUT1  = 6

    ' PropGFX Lite I/O Configuration
    PG_RXSPEED    = 256_000
    PG_RXPIN      = 24
    PG_TXPIN      = 25
    MYCLKVAL      = %00000010_00000000_00000000_00000000
    MYDIRAVAL     = %00000010_00000000_00000000_11111111
    MYPINOKVAL    = %00000001_00000000_00000000_00000000
    MYDATASHIFT   = 0

    'Debug LED Configuration
    'LED1          = 0
    'LED2          = 1
    'LED3          = 3

    ' Servo Configuration
    SERVO1        = -1
    SERVO2        = -1
    SERVO3        = -1
    SERVO4        = -1
    SERVO5        = -1
    SERVO6        = -1
    SERVO7        = -1

    ' ENC28J60 Ethernet settings
    ENC1_SCK      = 4
    ENC1_SI       = 5
    ENC1_SO       = 6
    ENC1_INT      = 7
    IP_ADDR       = 192 + 168<<8 + 1<<16 + 200<<24

    'HM55B Compass Module
    HM55BEna      = -1    ' HM55B Enable (active low)
    HM55BClk      = -1    ' HM55B Clock (active neg edge)
    HM55BDI       = -1    ' HM55B Data input
    HM55BDO       = -1    ' HM55B Data output

PUB Start
```

pProp040

pProp040 is a new design in the pPropQL series, based around a MC68(LC)040 and a Parallax Propeller. (While it is not intended to run any QL software per-se nothing impedes it either. A reconfiguration of the memory map may be needed.)
This system builds on the experience gained through the other systems but using a even more powerful processor.

Goals

A single board computer able to run a variety of OSs. The availability of a MMU means that un*x derivatives could be used.

1. Main Board

MC68040 Processor, intended frequency is 25 to 33 MHz (depending on processor available).
Up to 16MB (x32) Static RAM with 2 cycle access (fastest) and 2-1-1-1 burst access (8 MB per board, two boards can be stacked for 16 MB).
Parallax Propeller as boot device and slow text video interface (for testing and prototyping).
32 bit Video subsystem. 32 bit Video subsystem shares memory with the processor using BG/BR/BB protocol, synchronous to the processor clock, so 25 MHz and 33 MHz for 640x480 or 800x600.

1.1 Design

The 5 V MC68040 is interfaced to 5V memory and to a 5V tolerant CPLD. The CPLD runs all the glue logic and decoding needed to map memory and to guaranty fast memory access. The '040 does not have dynamic bus sizer thus a extra FPGA (the video controller) will act as one. Level shifting is only needed for the propeller, connected to the FPGA and the SD cand also connected to the FPGA.

(Partial) Render of version i2 (sent to manufacturing)

The design show above has been sent for manufacturing and 4 prototypes have been made. Currently waiting for the 040s to arrive. The glue-logic has been partially written (Verilog) and tested.

pPropellerSim

It is a Parallax Propeller Simulator with a twist it allows you to edit, compile and debug your assembler code. In that sense it is not a complete simulator, but has enough to let you program comfortably.

Note: This program requires java v1.5 or better to run. If you want to make it run under java 1.4, just remove the generics used in FormCloseEvent class, java.util.Vector refs and related. That was added to avoid the v1.5 compiler's complaints.

It is assumed through this manual that the reader is familiar with the Propeller terminology. If this is not the case, please read the Propeller Manual first.

Features

- Display a disassembled view of COG instructions
- Single-step, etc debugging capabilities
- Load .binary files (Spin instructions are discarded) directly to the first cog's memory
- Compiler / Editor

Parallax syntax (what else)
_ as number separator
equates/defines/=
Built-in editor, assembler source files can be loaded, saved, compiled and debugged.
.binary files can be edited (copy source). Symbols are shown in assembler view, but not labels

- Memory dump uses symbols and highlight recently modified long (when in screen), no undo/redo
- Binary files, plain binary without headers, can be loaded and saved (always saves 496 longs)
- Breakpoints on Read/write or execute, accessible through a pop up menu in the Memory Dump

Assembler View

This panel shows the contents of COG 0's memory as disassembled instructions.
Labels and symbols are used for every memory (in COG's address space) accessing instruction.
This symbols are either generated when a binary file is loaded or taken from the recently compiled source file (in editor).

The buttons in the Toolbar (run, stop, step-over, step-into, run-to-cursor and compile), and the commands with the same names on the Run menu have the same actions:

- Run : Starts the cog at the current **PC** till a breakpoint is reached or the program reaches

address 496. Display is updated every 100,000 cycles (approx).

- Stop : Stops the current running program if any (at 100,000 cycles boundaries).
- Step-over : Runs till next instruction is reached, useful for subroutines (call instructions).
- Step-into : Runs just one instruction
- Run-to-cursor : Runs till the **PC** reaches the position of the cursor
- Warm reset : Resets the cog. **PC** and **C** and **Z** flags are reset (0, cleared and cleared, respectively).

It does not erase memory.

- Cold reset : reset the Hub, and COG(s). Erases all memory and **PC** and **C** and **Z** flags
- Compile : Compiles the current editor file to COG's memory

Simulator

The included simulator tries to simulate cycle-exact operation (if that it is possible). So what you would expect in a real propeller (between the limitations described) should be reproduced here. Self-modifying code has the same caveats as a real Propeller does.

The Propeller works with a sort of 4 stage machine. (As described by Paul Baker).

- Source fetch
- Destination Fetch
- Execution
- Write-back

Instructions that need more than 4 cycles are delayed at the execution stage, well is what I believe, and what I implemented. Next instruction fetch occurs at the execution stage. So Self-modifying code, can not successfully modify the next instruction and execute the new modified instruction if there is no other instruction in-between because the old instruction was already fetched. Jump instructions some-how
avoid a "delay-slot", fetching the destination during source fetch, I did it also in the execution stage.

The Hub synchronization scheme is also implemented, so synchronization is needed to complete HUB instructions. Enough wait states are used till the Hub is synced to the Execution stage (where I believe the sync occurs).

Several threads at Parallax Forums describe part of the behavior implemented here, the rest, was gess-work. I did not peek at GEAR so I do not know how it was implemented there.

Any new info is of course welcome.

Breakpoints

A breakpoint can be set/reset at he current SELECTED row by tapping with the right mouse button (CTRL-tap in one button mice) to pop-up the Breakpoint's menu, in the assembler view pane.
The options are:

- Set breakpoint : Sets an execution breakpoint.
- Set read breakpoint : Sets a breakpoint for a read instruction on this address
- Set write breakpoint : Sets a breakpoint for a write instruction on this address
- Set breakpoint (eq) : Sets a breakpoint that is going to be triggered when the contents

of this address is equals to the supplied value.

- Set breakpoint (ne) : Sets a breakpoint that is going to be triggered when the contents

of this address is not equal to the supplied value.

- Clear breakpoint : Clears all breakpoints for this address.
- Clear all breakpoints : Clears all breakpoints for all addresses

Some breakpoints are exclusive, that means they cannot be set at the same address: eq and ne.

Registers View

This panel shows the current state of hardware registers, **CNT**, **PC** and flags. Newly modified registers are red marked. To change the status flag you can simply tap with the mouse button over it. The **PC** can be modified with the provided field, with immediate fetch of new instruction.

COG Memory Dump

This panel allows to view the current cog's memory contents and to modify them.

Value Modification

To modify the contents of a long, just tap over its value, and an editing field will be shown with the current value. An HEXADECIMAL value between 0 and ffffffff is expected. No '$' symbol is needed. Optionally an assembler instruction can be entered, in the same form as in the editor. Current symbols can be used for source/destination fields, but not symbols will be created during the compilation phase. An input will be accepted only when its contents are valid.

The trace column shown how many times a instruction was fetched, as a sort of profiling hint.

Hardware

- Not yet implemented

Editor

The editor is a simple text editor, used to edit source code (assembler) prior to be compiled.
The Compile button is used to compile this source and to load the compiled version directly into the cog's memory (starting always at address 0).
The cursor position is shown at the lower left position, starting at 1, 1.

Syntax highlight

The editor supports syntax highlight, so instructions, conditions, numbers, symbols and comments are displayed with unique colors for easy identification.

Compiler

The compiler is a simple-yet-straightforward two-pass assembler. Symbol creating as well as sizing take place on the first pass, while code is generated in the second pass. A listing file is created every time a compile is done. its name is the assembler file's name plus ".lst".

Errors

Detected errors are going to be displayed in the status bar (lower left panel). Several errors can be detected, the culprit is normally shown between parentheses, with the line number:

Compilation successful : No errors detected.

- Duplicated symbol : the symbol was defined more than once.
- Condition unknown : the condition was not recognized, probably was misspelled.
- Instruction unknown : the instruction was not recognized, probably was misspelled.
- Symbol not found : A symbol was used, without being defined.
- Duplicated wz or wc field : The modifier (effect) was present more than once.
- Argument(s) missing : An instruction requiring one or two arguments are missing one or both.
- Destination read-only : An attempt to use a read-only register (between 0x1f0 and 0x1f3) as

destination was found

- Garbage at end of line : Extra not needed arguments, not properly quoted comments, etc

(garbage), was found at the end of a line.

Syntax

Normal Propeller Tool syntax is supported with some caveats:

- No ORG directive is supported, not needed, code always starts at 0.
- No RES meta command is supported, nor it has any use at this point, use long 0 instead.
- No multi-line-comments are supported, just single lines with '

The rest should work without problems.

Note: A call missing the corresponding 'xx_ret' symbol is going to give a "Symbol not found error".

Load/Save

Programs (in text form, i.e. not compiled) can be loaded/saved with ctrl-o/ctrl-s or from the
File menu with ease. The default (fixed) invented extension is '.pasm'.

Hub Memory Dump

This frame provides a rapid way of peeking at the contents of the Hub memory.
Included functions are load/save the contents to a binary file ('.bin'), search and fill.

Search

The search button allows to search for the text/pattern in the search field.
An ASCII text can be entered directly, up to 24 characters are going to be used.
A HEX pattern can be entered when it begins with a '$' sign. Just one '$' is needed.
Hexa characters are composed of one or two Hex digits, separated by spaces, or in the case of only two-digit hex values, no spaces:

$12 34 56 is the same as $123456, usw.

Fill

This function allows to fill the (writable) area of the hub's memory with a defined pattern.

Options are:

- Random values: pseudo random values are used
- Incremental value : a counter from 0 to 255 is used
- Pattern : with the same value
- Clear all memory : It clears (writes with zero) the hub's memory
- Apply : applies the operations
- Cancel : dismisses the dialog

Keys, Commands and menus

File Menu

CTRL-N New : Cold resets the Propeller (does not clear the editor)
Open Propeller tool binary : Opens a .binary file in the assembler view, ram of cog 0 (compiled with Propeller Tool)
Open binary : Opens a file as binary in the assembler view, ram of cog 0 (compiled with this pPorpellerSim).

Save binary : Saves the contents of the cog 0 ram as binary
CTRL-O Open assembler : Opens a text file (.pasm extension) in the editor, previous content is lost without warning, cog 0 memory is unaltered
CTRL-S Save assembly : Saves the contents of the editor as a text file (.pasm etension)
CTRL-Q Exits the pPropellerTool

Recently opened files are added to the Recent open files menu

Edit Menu

Normal Undo, Copy, Paste stuff.
Copy to Editor : Disassembles the contents of the cog 0 memory (show in
assembler view and memory dump) and transfers it to the
editor. Previous contents of the editor are lost without
warning.
CTRL-UP Cursor Up Assembler view cursor movement, one row up
CTRL-DOWN Cursor down Assembler view cursor movement, one row down

Configuration Opens config Panel, usual color stuff and
the switch to turn on/off autosave of source file
before compilation
Run Menu

F10 Compile : Compiles the contents of the editor and transfers them
to the cog 0 ram. Previous contents of the cog 0 ram are
lost without warning. Symbols are preserved and used in
the assembler view.

F5 Run : Runs the contents of cog 0 ram starting at the current PC
if Address 496 (PAR) is reached, execution stops
F4 Run to cursor Runs the contents of cog 0 ram starting at the current PC
till the position of the current cursor
F8 Step over Runs till next instruction is reached starting at the
current PC
F7 Step into Runs current instruction, goes into CALLs

Data Menu

Hub Memory Dump Displays the Hub memory dump frame
Program Device Placeholder for when the programming of
propellers works
Copy HUB RAM to COG RAM allows to copy up to 496 longs from HUB RAM
back to COG RAM.

Known bugs

- HUB instructions, besides read and write the rest are not (yet) implemented

Propeller programming

NOTE: this was removed because... it does not work :-(

In order for pPropellerSim to be able to program Propeller chips, some libraries and
java extensions should be installed in your computer, i.e. RXTX and Sun Java Comm extension.

http://java.sun.com/products/javacomm

For GNU/Linux, I'd suggest you visit:

www.rxtx.org

For MacOS X, A good source of files and information is available at:

http://www.uow.edu.au/~phillip/MacInOut/serial.html

Note: is important that the directory /var/lock can be written by the members of
the group uucp, change permissions and ownership to reflect that!

For Winblows users, there is some info at www.rxtx.org, but I did not test it.

In the Programming Device Frame, (Data menu -> Program device), select the port
where the programming dongle resides. (Under linux choose something like /dev/ttyUSB0)
and for MacOS X choose /dev/tty.usbserialxxxxx.

Verify that a Propeller can be found on that port pressing the button "Discover".

Now that the propeller was found, you can program it (you can avoid the discover step
if you know for sure that a Propeller dongle/Propeller chip is connected and they work).

From the panel on the left choose the source, a PropellerTool compiled .binary file or
the current contents of the HUB memory. Press the button "Program" and if everything
goes without problems, your device should be programmed in few seconds. Any errors
found will be promptly reported.

Assembler compiler

The compiler has been rewritten from previous version (starting at 0.7.x) to match the features of the pLMMAss (pacito LMM Assembler).

Sections (.section)

Sections separate symbol creation into different groups

HUB : flat address space, addresses have a byte granularity
COG : flat, used for code that will be loaded in to a cog's memory,
addresses have a long granularity (and symbols will start at
adress 0 after the section .directive)
LMM : addresses have a long granularity

Alignment (.align

pPropQL

A hardware emulator of the SinclairQL using the propeller. For a MC68020 based one see pPropQL020

Note: The original thread at Parallax' forums is here

Introduction

The SinclairQL launched to the market in 1984 is a MC68008 based computer with custom supporting chips. The simplicity of its design compared to the Amiga or the AtariST make it a suitable candidate to what I call hardware emulator. Using an original processor and replacing the logic and custom chips with some glue logic and 2 propellers it is possible to get a replica. The functionality provided via two propellers, one as video controller and the other one as IO controller is enough, in my humble opinion, to get a functioning QL without using a real QL using any of the ROMS available (JS, JM, Minerva).

The aim of this project is to recreate this machine to the point that unmodified firmware and software can be used in it. Two prototypes have been built so far but more could be built if desired/needed.

pPropQl first prototype

Interfacing the propeller to a 5V MC68008

Level shifting

The MC68008 is a 16/32 bit processor, the smallest in the M68K family, with an external 8 bit BUS. It has a 20 bit address space in its 48 pin version and 22 in its 52 PLCC variant. It was fabricated in a NMOS process and requires 5V to operate properly. At the time of designing of the original QL 5V systems where common but today they have been passed out by lower voltage and less power hungry systems. The propeller cannot withstand a 5V signal, it has no 5V tolerant inputs. So several ways of interface exist. The simplest, but not necessarily the best, is to use a current limiting resistor. The potential will be limited by the protection diode incorporated to the inputs. A better and recommended method is to use a level shifter.

8 bit bus

The MC68008 has one bus cycle every 4 clock ticks like the MC68000 but it only has an 8 bit bus so two bus cycles are needed to read instructions limiting the maximum throughput to 1 MIPS @ 8MHz. Despite that the original QL had a cycle-stealing video controller, this design has a mirrored video RAM (mirrored to the HUB RAM) and thus affords faster execution times.

The bus is shared between the memory, the processor and the 2 propellers. The propellers are connected using current-limiting resistors and the bus is buffered using a 74HCT245 8 bit buffer.

Address bus

The address bus that arrives to the propeller is multiplexed using a pair of HCT157 and controlled with the propeller. As only one propeller will access the BUS at a time the two controlling signals (HL0 and HL1) are level-shifted using a pair of tri-state buffers (HCT125).

Memory organization

The QL has a simple memory map and pPropQL is hardwired to it.

Memory region	Use
0x00000-0x0BFFF	ROM/EPROM
0x0C000-0x0FFFF	External ROM
0x10000-0x17FFF	Unused
0x18000-0x1BFFF	I/O
0x1C000-0x1FFFF	External I/O
0x20000-0x27FFF	Video RAM first screen
0x28000-0x2FFFF	Video RAM second screen, used for system variables
0x30000-0x3FFFF	User RAM

Using some decoding logic this map can be easily implemented (This decoding logic was implemented in the board show above and in the code below, but it is contained in the CPLD, it remains here for easier understanding!).

An extension to this map is any RAM in the area 0x40000 to 0xFFFFF. The pPropQL provides one decoding signal for the upper 512Kbytes, those could be used for an EPROM (as provided) or for more RAM but a discontinuous area will exist. The code shown below will hold **NRAMCS** asserted (low) for the area between 0x20000 and 0x7FFFF. The area above is controlled by **UCS**.

All this glue logic can be put into a CPLD. Even the smallest with only 36 macrocells can do it (XC9536).

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Company:
// Engineer: Pacito.Sys
//
// Create Date:    09:04:29 01/21/2010
// Design Name: pPropQL glue logic
// Module Name:    gluelogic
// Project Name: pPropQL
// Target Devices: XC9536
// Tool versions: Xilinx WebISE 10.1
// Description: Glue logic for the pPropQL
//
// Dependencies:
//
// Revision:
// Revision 0.02 - Simulated
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module gluelogic(
    input in_clk, // clock input
    input in_fc0, // function code 0
    input in_fc1, // function code 1
    input in_nas, // Adress strobe (Asserted when low)
    input in_nds, // Data strobe (Asserted when low)
    input in_rw, // Read-write signal
    input A15, //
    input A16,
    input A17,
    input A18,
    input A19,
    output INTA, // Input to the uP, VMA
```

```
    output NDTACK, // DTACK, asserted when low
    output HL, // HL signal for the '157s
    output IOR, // IO read signal
    output IOW, // IO Write signal
    output VIDEOW, // Video W
    output NROMCS, // ROM CS
    output NRAMCS, // RAM CS
    output NOE, // OE for the RAM
    output W, // W for the RAM
    output PROMCS, // Extra ROMCS for propeller
    output UCS // Extra 80000..FFFFF chip select
    );

reg [3:0] r_dtackshift;
/* pPropQL Memory Map
 *
 * 00000..0FFFF RW NROMCS
 * 10000..17FFF RW -
 * 18000..1FFFF R  IOR (with waitstates)
 * 18000..1FFFF W  IOW (with waitstates)
 * 20000..27FFF W  VIDEOW, RAMCS (with waitstates)
 * 20000..27FFF R  RAMCS
 * 28000..7FFFF RW RAMCS
 */
// Internal signals
wire int_r = ~(in_nas | ~(in_fc0 & in_fc1)) | ~in_rw | in_nds;
wire int_w = ~(in_nas | ~(in_fc0 & in_fc1)) | in_rw | in_nds;
wire int_romcs = A19 | A18 | A17 | A16;
wire int_extra = A19 | A18 | A17 | ~A16 | A15;
wire int_io = A19 | A18 | A17 | ~A16 | ~A15;
wire int_video = A19 | A18 | ~A17 | A16 | A15;
wire int_ramcs = A19 | ~int_romcs | ~int_io | ~int_extra;

// ** Outputs **
assign INTA = in_nas | ~(in_fc0 & in_fc1);
// Chip selects
assign NROMCS = int_romcs | int_r;
assign PROMCS = int_romcs | int_r;
assign NRAMCS = int_ramcs | in_nds;
assign VIDEOW = int_video | int_w;
assign IOR = int_r | int_io;
assign IOW = int_w | int_io;
assign UCS = ~A19 | in_nds;
// Extra outputs
assign NOE = int_r;
assign W = int_w;
```

```
// DTACK generation

wire int_slow = VIDEOW & IOR & IOW;
assign NDTACK = (r_dtackshift < 14) & ~int_slow;
// High, low signal
assign HL = r_dtackshift > 1;
always @(posedge in_clk)
begin
    if (in_nas)
        r_dtackshift <= 0;
    else
        if (~int_slow) begin
            r_dtackshift <= r_dtackshift + 1;
        end
end

initial
begin
    r_dtackshift = 0;
end
endmodule
```

Report:

Macrocells	16/36 (45%)
Pterms	56/180 (32%)
Registers	4/36 (12%)
Pins	23/34 (68%)
Function blocks	26/72 (37%)

Interfacing to software emulated peripherals

The fun part is actually the implementation of all I/O by means of two Propellers. One for video (32k RAM used) and the other one for the rest. The microdrives are replaced with a SD card. The keyboard is emulated using a PS/2 keyboard, the RTC is a DS1307 and one serial port is also provided.

As the 68K has an asynchronous bus, the bus termination signals are generated by the video propeller and level-shifted using a HCT125 tri-state buffer. This circuit defaults to 4 clock bus cycles for fastest memory access. The Video RAM is mirrored to one of the propellers and thus these write (but not read) accesses can also be slower.
Note: Due to software development it has been noted that 500 ns for I/O access is not enough. The **DTACK** signal is generated then using a CPLD with code shown above.

To successfully interface to the 68K for I/O, the propeller has to latch the address bus, read or write onto the data bus between the stipulated time. For sake of argument let's think that a time of 500 ns is enough.

Asynchronous write cycle. Note that **DTACK** has to be valid before the transition S4->S5. The code below only uses the low part of the address bus and does not touch the **DTACK** signal, because it is generated automatically.

```
IOR             mov       DIRA, #0

ior_loop        waitpne   c0_IOR, c0_IOR
                or        DIRA, c0_SHIADDR        ' Selects low address
(0)
                mov       c0_addrl, INA
                shr       c0_addrl, #8
                and       c0_addrl, #255          ' low address

                rdbyte    OUTA, c0_addrl          ' reads data
                or        DIRA, #DATABUS          ' DATA bus is output
                nop
                nop
                nop
                mov       DIRA, c0_dira_def       ' Data bus as INPUTS
                jmp       #ior_loop               ' restarts loop

c0_SHIADDR      long      %000_0_0_0_0_1_00000000_00000000_00000000 ' Wh
en high high address
```

In the example above a very simple mechanism where one COG answers the 68K with what is available in HUB RAM in the first 256 bytes. The QL only uses a few ports in this 32kbytes area, so all the address decode is not necessary. Housekeeping COGs are going to fill the HUB with the right values when an event occurs. Another method is to perform all the tasks, i.e. housekeeping, within the answering COG delaying the processor.

Video

One of the two propellers acts as a video controller. The QL has only 2 modes in its standard form. 256x256 8 colors and 512x256 4 colors. An image of a working driver for 512x256 4 colors is shown below.

At this point the image is 512x256 pixels in a 640x480 frame with a 25 MHz clock. The unused area is regarded as border.

ROM

The goal is to use an unmodified JS or JM ROM. A Minerva ROM could also be used. In the sense of speeding up few parts of the IO, some modifications could be implemented. Specially slow is the access to the i8048, keyboard controller among other things.

pPropQL020 by Pacito.Sys

An extended version of pPropQL. As MC68008 are scarce and I have some, plenty, MC68EC020s I decided to improve pPropQL with a more powerful processor, more memory, programmable logic and loads of fun.

The glue logic of pPropQL has been incorporated into three small CPLDs, they could be replaced by a bigger, and I mean with more IOs, CPLD. The rationale behind the CPLDs is: this board can be re-targeted. The memory map of the QL is just not that nice to run for instance OMU or even uCLinux (M68K port).

The two I/O Propellers have been kept but now the control over the address mux is handled by the memory-decoding logic and a wait-state generator, as well as the generation of the DSACK0/1 (bus termination signals), freeing powerful propeller pins.

The EPROM that existed in the pPropQL has been left out, so booting and loading a kernel image is the task of a boot-loader inside the video Propeller. This bootloader reads the the image from a SD/MMC card, copies it to memory and executes it.

Note: assert and negate are used to specify forcing a signal to a particular state. In particular, assertion and assert refer to a signal that is active or true; negation and negate indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.
Note: Signals are in **bold**. A signal's name starting with an N means that that signal is active low, assertion will mean a logic 0 level in positive logic (used here). Only one of the three will preferably used here.

A discussion of this board can be found here.

pPropQL020 Prototype pPropQL020 with 2 MB SRAM soldered

Hardware description

Level shifting

The MC68EC020 processor is a 5V part thus some level shifting is needed between the propeller and the 68K. The address bus mux, inside the third CPLD, acts as a level shifter. Its core voltage is kept at 5V while its IO voltage is 3.3V. This part is a XC9536.
The data bus is buffered via a 74ABT245 and connected to the propeller using current-limiting resistors. Control lines use also current-limiting resistors.

Glue logic

All glue logic has been implemented using XC9536s (CPLD). Memory decoding is handled by U11 and the glue is by U10. The Glue logic has to generate the byte strobe signals for memory and several other signals like **NHALT**, **NRESET**, **NOE** usw.
The 020 uses two Data Acknowledge signals to signal the end of the bus cycle and thus to determine the port size. The port sizes used are 32 bits for SRAM and 8 bits for the propellers. At the beginning of a cycle the signals **A1**, **A0**, **SIZ0** and **SIZ1** say what kind of transfer size is being performed and to which address. Decoding of those signals yields the byte strobes for writing (**NWE0** to **NWE3**).
In a 32 bit port, like the SRAM here, all byte strobes are used.

Byte strobe	bits used	Equations
NWE3	D7..D0	!(!A0 & !A1 & !RW & !NDS)
NWE2	D16..D8	!((SIZ1 & !A1 & !RW & !NDS) | (!SIZ0 & !A1 & !RW & !NDS) | (A0 & !A1 & !RW & !NDS))
NWE1	D23..D16	!((!A0 & A1 & !RW & !NDS) | (SIZ1 & SIZ0 & !A1 & !RW & !NDS) | (!SIZ1 & !SIZ0 & !A1 & !RW & !NDS) | (!SIZ0 & A0 & !A1 & !RW & !NDS))
NWE0	D31..D24	!((SIZ1 & A1 & !RW & !NDS) | (!SIZ1 & !SIZ0 & !RW & !NDS) | (A0 & A1 & !RW & !NDS) | (SIZ1 & SIZ0 & A0 & !RW & !NDS))

Control signal's relationship for the '020

The QL uses only **IPL2** as interrupt input and asserts the **NVMA** input in the MC68008 for autovector interrupts. On the MC68020 this **NVMA** signal does not exist but an **NAVEC** input for auto-vector interrupt recognition is provided. This signal is asserted when the function code outputs (**FC2** to **FC0**) signal an interrupt acknowledge cycle.

The DSACK generator for **DSACK0**..**1** uses a pair of cascaded shift registers. **HL** is asserted three cycles

after **NAS** when **NSLOW** is also asserted and thus the BUS cycle is not terminated by a 32-bit port access (i.e. both **DSACK0** and **DSACK1** are asserted before S4). This signal handles the address mux, contained in U14. After one of the IO strobes is asserted at least 50 ns pass before the propeller reads the low address byte so 3 wait states are needed as a minimum. After this read there is place for 2 or 3 more propeller instructions before the high address byte can be read. As the low address byte has to be masked and shifted this does not pose a problem. (See the logic analyzer's traces below).

Memory decoding is done with simple 3-to-8 decoders and some gates

The memory map used here is similar to the QL's except for the fact that the SRAM is mapped to the addresses 0x20_0000 to 0x5F_FFFF, 4 MBytes.

As I'm not really interested in using this for QL and more in porting OMU to it, this memory configuration maximizes available memory.

Video memory is as the built prototypes only writable but that can be changed to read/write with a simple piece of wire and a change to U11 to make VIDEOR available.

A Verilog implementation of the above circuits, both condensed into one CPLD could be this one. **EIOR** and **EIOW** are not implemented due to the lack of pins in the XC9536 used. A XC9572 could be used instead, then the Address/Data MUX could be fitted inside too.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Company:
// Engineer: Pacito.Sys
//
// Create Date:    09:04:29 01/21/2010
// Design Name: pPropQL020 glue logic
// Module Name:    gluelogic
// Project Name: pPropQL020
// Target Devices: XC9536
// Tool versions: Xilinx WebISE 10.1
// Description: Glue logic for the pPropQL
//
// Dependencies:
//
// Revision:
// Revision 0.02 - Simulated
```

```
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module gluelogic(
    input in_clk,
    input in_fc0,
    input in_fc1,
    input in_nas,
    input in_nds,
     input in_rw,
     input in_siz0,
     input in_siz1,
     input A0,
     input A1,
    input A15,
    input A16,
    input A17,
    input A18,
    input A19,
     input A20,
     input A21,
     input A22,
     input A23,
     output AVEC,
    output NDSACK0,
     output NDSACK1,
    output HL,
    output IOR,
    output IOW,
    output VIDEOW,
    output NROMCS,
    output NRAMCS0,
     output NRAMCS1,
     output NOE,
     output WE3,
     output WE2,
     output WE1,
     output WE0//,
     //output EIOR,
     //output EIOW
    );

reg [3:0] r_dtackshift; // wait state counter
/* pPropQL Memory Map
 *
```

```
 * 000000..00FFFF RW NROMCS
 * 010000..017FFF R  EIOR (with wait states)
 * 010000..017FFF W  EIOW (with wait states)
 * 018000..01FFFF R  IOR (with wait states)
 * 018000..01FFFF W  IOW (with wait states)
 * 020000..027FFF W  VIDEOW, NRAMCS0 (with wait states)
 * 020000..027FFF R  NRAMCS0
 * 1FFFFF..1FFFFF RW NRAMCS0
 * 200000..3FFFFF RW NRAMCS1
 */
// Internal signals
wire int_r = ~(in_nas | ~(in_fc0 & in_fc1)) | ~in_rw | in_nds;
wire int_w = ~(in_nas | ~(in_fc0 & in_fc1)) | in_rw | in_nds;
wire int_00 = A23 | A22 | A21 | A20;
wire int_romcs = int_00 | A19 | A18 | A17 | A16;
wire int_extra = int_00 | A19 | A18 | A17 | ~A16 | A15;
wire int_io = int_00 | A19 | A18 | A17 | ~A16 | ~A15;
wire int_video = int_00 | A19 | A18 | ~A17 | A16 | A15;
wire int_nramcs0 = A23 | A22 | A21 | ~int_romcs | ~int_io | ~int_extra
;
wire int_nramcs1 = A23 | A22 | ~A21;

// ** Outputs **
assign AVEC = in_nas | ~(in_fc0 & in_fc1);
// Chip selects
assign NROMCS = int_romcs | int_r;
assign NRAMCS0 = int_nramcs0 | in_nds;
assign NRAMCS1 = int_nramcs1 | in_nds;
assign VIDEOW = int_video | int_w;
//assign EIOR = int_r | int_extra;
//assign EIOW = int_w | int_extra;
assign IOR = int_r | int_io;
assign IOW = int_w | int_io;

// Extra outputs
assign NOE = int_r;
assign WE3 = ~(~A0 & ~A1 & ~in_rw & ~in_nds);
assign WE2 = ~( (in_siz1 & ~A1 & ~in_rw & ~in_nds) |
                (~in_siz0 & ~A1 & ~in_rw & ~in_nds) |
                     (A0 & ~A1 & ~in_rw & ~in_nds) );
assign WE1 = ~( (~A0 & A1 & ~in_rw & ~in_nds) |
                     (in_siz0 & in_siz1 & ~A1 & ~in_rw & ~in_nds) |
                     (~in_siz1 & ~in_siz0 & ~A1 & ~in_rw & ~in_nds) |
                     (~in_siz0 & A0 & ~in_rw & ~in_nds) );
assign WE0 = ~( (in_siz1 & A1 & ~in_rw & ~in_nds) |
                     (~in_siz1 & ~in_siz0 & ~in_rw & ~in_nds) |
```

```
                    (A0 & A1 & ~in_rw & ~in_nds) |
                    (in_siz1 & in_siz0 & A0 & ~in_rw & ~in_nds) );

// DTACK generation

wire int_slow = VIDEOW & IOR & IOW;
assign NDSACK0 = (r_dtackshift < 14) & ~int_slow;
assign NDSACK1 = int_slow;

// High, low signal
assign HL = r_dtackshift > 1;
always @(posedge in_clk)
begin
    if (in_nas)
        r_dtackshift <= 0;
    else
        if (~int_slow) begin
            r_dtackshift <= r_dtackshift + 1;
        end
end

initial
begin
    r_dtackshift = 0;
end
endmodule
```

Propellers as peripherals

Two propellers are used as video controller one and IO controller the other one. They are memory mapped and thus act as any other memory. This propellers present a simple interface, data bus (**D24**..**31**) address bus (multiplexed A0 to A15) and one strobe signal per function, read, write and so on.

Upon assertion of the strobe signal the propeller decodes the address and performs whatever action is needed. All this processing has to take place in the allocated time, 15 or 16 MC68K cycles. 1 us @ 16 MHz or 2 us @ 8 MHz (used now for testing).

The following logic analyzers' trace show the state of various signals when NIOR is asserted. **PA0**..**7** are the multiplexed addresses and **D24**..**31** are the lowest significant bits of the data bus used for 8 bit transfers.

The assertion of **NVIDEOW** proceeds in a similar manner.

As can be seen the data bus changes state some 700 to 800 ns after the assertion of **NIOR/NVIDEOW**. As the data is latched on the rising edge of **NDS** (or'd to **NIOx/NVIDEO**) there is enough time. All this examples were done @ 8 MHz.

The answering code for this could be something like this:

```
DAT
                        org     $0

VIDEOCOG                mov     DIRA, #0

c2_videoemu             waitpne c2_c_VIDEOW, c2_c_VIDEOW ' waits for V
IDEOW to be asserted

                        mov     c2_v_addr, INA          '@ 40   gets l
ow part of address
                        shr     c2_v_addr, #16          '@ 80
                        and     c2_v_addr, #255         '@120
                        add     c2_v_addr, PAR          '@160 adds vid
eo buffer offset
#ifdef M68K_8MHz
                        nop                             ' this NOPs en
sure that the high address
                        nop                             ' is available
 to be read
                        nop
#endif
                        mov     c2_v_addrh, INA         '@320 now it i
s safe to get high addr
                        mov     c2_v_data, c2_v_addrh
                        shr     c2_v_addrh, #8          '@360
                        and     c2_v_addrh, c2_c_MSKADDR
                        add     c2_v_addr, c2_v_addrh   '@400
                        wrbyte  c2_v_data, c2_v_addr    '@440
                                                        '@715 (max)
                        waitpeq c2_c_VIDEOW, c2_c_VIDEOW ' waits for N
VIDEOW to be negated
                        jmp     #c2_videoemu


c2_c_VIDEOW             long    1<<VIDEOW
```

```
c2_v_addr               long    0
c2_v_addrh              long    0
c2_v_data               long    0
c2_c_MSKADDR            long    $00000f00   ' only 4 kbytes !!!!
```

Reset

The reset is handled by one COG and one PIN as output. As the video propeller acts as boot ROM, it answers the **NPROMCS** signal. After reset the stack pointer and the reset vector are fetched and execution starts at that address (0x0000_0010 in this case).

The following code fragment shows the (wasted) cog. Any COG would do in principle, so it can be devoted to something else afterwards.

```
DAT
                        org     $0

RESETCOG                mov     OUTA, #0
                        mov     DIRA, c1_c_DIRA

                        or      OUTA, c1_c_RESET        ' negates NRES
ET and NHALT
                        mov     c1_v_wait, CNT
                        add     c1_v_wait, c1_c_80M
c1_loop                 waitcnt c1_v_wait, c1_c_80M
                        jmp     #c1_loop

c1_c_DIRA               long    1<<RESET           ' RESET
c1_c_RESET              long    1<<RESET
c1_v_wait               long    0
c1_c_80M                long    80_000_000
```

Programming in C

Currently there are no C compilers available for the Propeller. But progress is being made in this area.

ImageCraft ICCv7 for Propeller C STD

ImageCraft are currently developing a Propeller target for their well established range of embedded C compiler tools. Tentative price is $199. The product includes a Windows IDE, C compiler, relocatable assembler, linker and other tools. (The implication being that there will not be any Mac or Linux support). It will support the Large Memory Model (LMM).
April 19th 2007 Announcement.
Nov 17th 2007 Status Update.
Dec 19th 2007 Status Update reads:

- *Compiler:*
  - *All non-floating point operations supported*
  - *need to do: function entry / exit code*
  - *need to do: register allocator tuning, peephole, codegen optimizations*
- *Assembler:*
  - *all instructions, effects etc. supported*
  - *most directives supported. Memory area attributes need to be done*
  - *operators not yet supported*
- *Linker:*
  - *Most relocations done*
  - *need to support library files*
- *Library*
  - *at the minimum, need to support mul and div/mod*
- *C machine LMM kernel*
  - *Design mostly complete, coding to be started*
- *IDE*
  - *not yet started, need Propeller download utility*

Others

Some people have suggested porting one of the existing open source GPL C compilers: GCC (Gnu C Compiler) or SDCC (Small Device C Compiler). Both these were written to be retargetable, so a back end for the Propeller should be possible, again probably using the LMM. So far no one has announced that

they are working on this. An open source LMM assembler is probably a prerequisite, and as yet no one has released one.

Catalina - a FREE C Compiler for the Propeller Chip

Ross Higson has developed Catalina - a free ANSI compliant C compiler for the Propeller Chip. Catalina can be downloaded from SourceForge. The current release is 3.0.

Catalina is based upon LCC (a robust, widely used and portable C compiler front-end), with a custom back-end that generates Large Memory Model (LMM) PASM code for the Propeller. For general details on LMM, go here.

Catalina is now essentially complete. Bugs and maintenance releases will still be issued, but no more functional additions are expected - at least not until the Prop II arrives!

The Parallax forums should be used for contacting the author or interacting with other Catalna users - see this Parallax forum thread

Major features of Catalina:

- ANSI C compliant, with C89 library (plus some C99 library functions).
- Floating point support (32 bit IEEE 754).
- Debugger support. Several debuggers are now supported:
  - POD debugger and PropTerminal for assembly level debugging; or
  - **BlackBox** (Windows or Linux command line) for source level debuggin; or
  - BlackCat (Windows GUI) for source level debugging.
- Basic support for **all** Propeller platforms, plus extended support for specific platforms - the targets provided support the **Hydra**, **Hybrid**, **TriBladeProp**, **Morpheus**, **DracBlade**, **RamBlade**, **C3** and the **Demo** Board. Others can be easily added.
- Platform independent - **Win32** and **Linux** binaries provided. Easily ported to other platforms.
- All source code provided (compiler, libraries and tools).
- XMM support for programs larger than 32K (requires additional hardware).
- Supports the **Code::Blocks** Integrated Development Environment.
- **It's FREE!**

Catalina supports three different addressing modes:

- Tiny - all code and data share the 32Kb of Hub RAM. This mode is used by all LMM and EMM programs, and is suitable for use on any Propeller platform.
- Small - code can be up to 16Mb, but all data (including the stack and heap) must still share the 32kb of Hub RAM. This is the original XMM mode as implemented in the various beta releases. This mode requires dedicated XMM hardware (i.e. external SRAM). Currently supported are the **Hydra** and **Hybrid** (using the HX512 external SRAM card), the **TriBladeProp**, **RamBlade**, **DracBlade Morpheus** and **C3**.
- Large - code, data and heap can be up to 16Mb (combined), and only the stack uses the 32Kb of Hub RAM. This mode uses a completely new code generator (the previous code generator remains

in use for the other modes), and also an enhanced XMM Kernel. This requires the same dedicated XMM hardware as the Small mode. When the Prop II eventually surfaces, the space available for stack under the Large addressing model is expected to be increased to 256Mb. However, note that 'larger' is not always 'better' - programs that use the larger addressing modes will generally be slower than programs that use the smaller addressing modes. A programs should always use the smallest addressing mode it can.

Other notable enhancements in recent releases are as follows:

- Multi-cog and multi-thread support. Catalina can execute C code on all 8 cogs concurrently, and execute multiple C threads on each cog.
- An SD Card program loader (**Catalyst**). Catalyst is an interactive program loader and a set of utility programs that simplifies the execution of Catalina programs (LMM and XMM) on platforms with an SD card. Catalyst can also be used to load normal SPIN/PASM binaries. Catalyst includes several example applications programs, such as the **vi** editor, the **Lua** scripting language and a fully ISO compliant **Pascal P5** compiler. These require XMM RAM.
- A serial program loader (**Payload**). Payload can load LMM or XMM programs (up to 16Mb) into the Propeller directly from a PC. Payload can also be used to load normal SPIN/PASM binaries.
- Suport for **Code::Blocks** for graphical editing and compiling of C programs. **Code::Blocks** can be downloaded from here
- Support for multi-CPU systems such as the TriBladeProp and Morpheus. This includes support for "proxy devices" (in a multi-CPU system, this is the ability to use devices physically connected to another CPU as if they were directly connected to the CPU on which the Catalina program is running).
- Simplified compiler command-line options (no more obscure options like **-Wl-W-d** or **-x5**). Environment variables can be used to to store commonly used configuration options.
- A standard target package that inlcudes all supported platforms. All platform-specific target configuration can now be done on the command-line using. New "custom" platforms can be added to the standard target package, or a completely new target can be created instead.

Here is a picture of Catalina being used from within **Code::Blocks**:

PropForth

Sal Sanci has revised SpinForth and posted PropForth. There is a spin template that appended with forth definitions. The spin file is loaded into eeprom as any spin program. After that, the propeller can support any serial terminal (teraterm was used successfully). The forth dictionary can be extended in ram using standard colon definitions. The RAM image can be resaved to eeprom using the prop specific "saveforth" word. New definitions are added at the end, redefinitions of words hide old definitions. Eventually, it becomes advantageous to re-generate the propforth image. The spinmaker word causes the prop to generate a new version of the forth source text which can then be copied and pasted into the propforth-template.spin file. This allows the Propeller Tool to resolve any forward references in the forth source code without the use of defered words. Regenerating the propforth.spin image alse recovers dictionary space lost to redifined words, as spinamaker only finds and uses the most recent definition for each word.
There is a software logic analyzer extension which can use 1 cog to sample all the i/o pins at a rate of 40 or more clocks, or uses four cogs to sample all the pins every clock.
There is default support for the forth software cooperative multitasking round robin, which transfers control from task to task using the pause word.
There is also an assembler multitasker that occurs between forth words. This permits very frequent access to whatever needs very frequent access. Since this has an impact on the speed of execution on the core, the assembler time slicer can be made to run on a specific core, and has no effect on the others.
The documentation is being revised for clarity. The order of the topics addressed is driven by questions that are asked.
The project can be found at
http://code.google.com/p/propforth/

PropellerForth

Cliffe Biffle has created an open source complete Forth development system for the Propeller, called PropellerForth. If you are not familiar with what Forth is about, its basically a programming language and development environment that runs entirely on the Propeller. You do not need a PC to program your Propeller board (Prop Demo, Hydra supported) anymore. Once you use the Propeller Tool to upload the single binary image all your other development tasks are performed on the Propeller directly. Simply plug in a keyboard, hook up a TV, and off you go! I was a little confused a while back about what Forth is really about, so the best analogy I can give would be: It's like an open programmable dynamic "operating system" that can be modified on the fly, in real-time, while the system is running.

Your best bet is to go download this rather amazing peice of work, and install it on your Propeller system. More information and links to file downloads available on the PropellerForth homepage.

It's important to note that this system is complete and ready to run right now. At the time this page was created, they were at version 8.01.

JDForth

On 8 August 2008, Carl Jacobs introduced a commercial Forth to Spin compiler for the Propeller, called JDForth. In contrast to PropellerForth (and most other Forths), this Forth is able to co-exist with Spin in the Propeller.

- JDForth allows for the easy inclusion of propeller assembly words. These words may be either linked at compile time, or dynamically included at run time.
- JDForth has words to support 32-bit floating point in IEEE-754 compliant format.

The details of JDForth - as well as purchasing information - can be found at the JDForth homepage.

JDForth was announced on the Parallax Forum.

A Propeller JVM for the Java Programming Language

Fast-Track

PropJavelin is a project to implement the functionality of the Parallax Javelin Stamp on the Propeller Chip. This is the implementation of a JVM which runs on the Propeller to allow Java(TM) programming of the Propeller. Java program development is undertaken using a modified version of the Javelin Stamp IDE.

A fast-track for getting the PropJavelin running can be found here

Introduction

Peter Verkaik proposed on the Propeller Forum to implement a JVM for the Propeller to enable the use of the Java(TM) programming language.

Original forum thread : here

End-user Java development would be undertaken using the Jikes(TM) compiler for the Java language and post processing by the JavelinDirect linker as used for Parallax JavelinStamp (suitably modified). Development could also be undertaken using the JavelinStamp IDE, to produce the required bytecode for inclusion with a Propeller JVM program or for download directly into a Propeller Chip running such a JVM program.

The bytecode generated by JavelinDirect is in a .jem bytecode file using JEM bytecode opcodes which have different values to Sun's JVM bytecode opcodes in a .class file although they are essentially the same.

Trademark Acknowledgements

Development Tools

The following development tools will be useful for anyone interested in becoming involved with the JVM

develoment.

Jikes Compiler

The Jikes compiler was developed by IBM and has been made open source with source code available under IBM's Public License.

The easiest way to get hold of the Jikes compiler executable for Windows is probably to download the JavelinStamp IDE from Parallax and install that. This will provide the graphical IDE for the JavelinStamp and allows the installation to be tested.

JavelinStamp IDE download page from Parallax : here

The Jikes compiler on Sourceforge : here

JavelinDirect / PropDirect Linker

Original JavelinDirect Source download page from Parallax : here

The updated JavelinDirect Package download via Yahoo Groups : here

A Propeller Specific JavelinDirect (PropDirect) hosted on the Parallax forum : here(forum thread)

JIDE Runtime

JavelinDirect and PropDirect require RTL60.BPL and VCL60.BPL to be installed in \Windows\System32.

The JIDE Runtime download via Yahoo Groups : here

The JIDE Runtime hosted on the Parallax forum : here(forum thread)

Example Compilation Process

Compiling a testJVM.java source file -

```
Set JIKESEXE="C:\Program Files\Parallax Inc\Javelin Stamp IDE\Jikes\Ji
kes.exe"
Set JIKESCLS="C:\Program Files\Parallax Inc\Javelin Stamp IDE\lib"

%JIKESEXE% -classpath %JIKESCLS% testJVM.java
```

Linking the testJVM.class to produce testJVM.jem and jem.out -

```
PropDirect --path .;%JIKESCLS% --link --debug testJVM
```

Including the testJVM.jem file in a Propeller Spin program -

```
PRI ShowAllBytecode_Version_1 | hubPtr
  repeat hubPtr from @JemBytecodeStart to @JemBytecodeEnd
    ShowThisBytecode( byte[ hubPtr ] )

PRI ShowAllBytecode_Version_2 | i
  repeat i from 0 to BytecodeLength - 1
    ShowThisBytecode( byte[ @jemBytecodeStart + i ] )

PRI BytecodeLength
  return @JemBytecodeEnd - @JemByteCodeStart

DAT
  JemByteCodeStart
  FILE = "testJVM.jem"
  JemByteCodeEnd
```

Links

Documentation

The Java(TM) Virtual Machine Specification Second Edition from Sun Microsystems, Inc.

Read online : here

Other JVMs which may be of interest

Java for GBA - A port of the KVM (J2ME/CLDC) for the Game Boy Advance.

Source Code : here

LeJOS/TinyVM - LeJOS is replacement firmware for LEGO Mindstorms RCX and NXT bricks, a continuation in development on its fore-runner, TinyVM.

LeJOS : here
TinyVM: here

NanoVM - Java for the AVR. The NanoVM is a Java virtual machine for the Atmel AVR ATmega8 CPU, the member of the AVR CPU family. The NanoVM and its tools are distributed under the GPL and can be used on other AVR based systems as well.

More information : here

SimpleRTJ - A small footprint Java VM for embedded and consumer devices. The simple Real-Time-Java for the small embedded and consumer devices is a clean room implementation of Java Virtual Machine that has been specifically designed to run on devices with the small amount of system memory. In contrast to other embedded implementations of the virtual machine the simpleRTJ requires on average about 18-24KB of code memory to run. The evaluation version of simpleRTJ for embedded and consumer devices is available under the RTJ Computing's non-commercial source code license for private, educational and evaluation purposes.

More information from RTJ Computing : here

Taurus Java VM - A clean-room Java™ Virtual Machine implementation, designed to be highly portable, optimised for use on small, low-power devices. From version 1.04a onwards, JavaVM is covered by the terms of the GNU General Public License (GPL).

Specification, Documentation, Whitepapers, Sources and Downloads.

More information from Taurus Software & Consulting : here

Development and Design Issues

The proposed Propeller JVM leverages the existing toolset used for development of programs for the JavelinStamp and the design of the Propeller JVM will therefore be influenced by that toolset.

The Jikes compiler is a full Java compatible compiler delivering complete bytecode for Java class files. The JavelinDirect / PropDirect tool bridges the divide between class files and the JEM bytecode the Propeller JVM will execute and thus is the primary tool (along with the JVM itself) which will influence design decisions and will need to change as design decisions are made.

Goal

There are two potential goals; a JVM which supports 'Java' as it is on the JavelinStamp (an emulation with just minor changes to the toolset to better suit the Propeller) or an enhancement to provide a Java which is ideally suited towards the Propeller with less regard to the JavelinStamp itself.

Development could be incremental and phased, delivering emulation then enhanced, or initially targeted to be ideally suited to the Propeller.

Primitive Type Size

Because the JavelinStamp is based upon a constrained architecture, int is implemented as 16-bit not 32-bit, long (64-bit) is not supported at all.

There are two choices for the Propeller, to retain the fundamental type-size as 16-bit internally with short and int as 16-bit and long as 32-bit, or to use a fundamental type-size of 32-bit with short as 16-bit, int as 32-bit and long as 32-bit rather than 64-bit.

The argument for a 16-bit type-size is less runtime memory use for holding method and local variables when a programmer uses int (16-bit) but that is set against a more complicated and larger foot-print JVM which has to support both 16-bit and 32-bit primitive data sizes.

The argument for a 32-bit type-size is the simplification and smaller foot-print of the JVM dealing with only one 32-bit primitive data size at the expense of higher memory use for method and local variables at runtime.

Virtual Peripherals

The JavelinStamp uses a 'Virtual Peripheral' mechanism to implement a degree of concurrent processing. Whether that should be retained for the Propeller or how it should be changed needs to be decided.

Native Methods

The JavelinStamp native methods are part and parcel of the Java class files provided as part of the Java programming environment. Which of those should be retained, changed or removed, and which new native methods should be added for the Propeller needs to be decided.

Threads

The JavelinStamp like many constrained JVM implementations does not support threads. While any JVM should ideally support threads it needs to be decided if the Propeller JVM will or not and if it does the JVM will need to be designed to support them.

Garbage Collection

The JavelinStamp like many constrained JVM implementations does not support garbage collection. While any JVM should ideally support garbage collection it needs to be decided if the Propeller JVM will or not and if it does the JVM will need to be designed to support them.

Pascal P4 for the Propeller Chip

AiChip Industries are developing a 'reference design' for running Pascal P4 derived bytecode on the Propeller Chip.

Original Parallax Forum thread : here

Overview

The AiChip_P4_XXX.spin program interprets P4 bytecode held in hub memory. That bytecode is produced by AiChip's PrettyP4 linker from the P4 p-code generated by a P4 compiler.

The P4 compiler written by Urs Ammann, Kesav Nori and Christian Jacobi, and described in the book "Pascal Implementation" by Steven Pemberton and Martin Daniels, produces P4 p-code. P4 p-code was designed for execution on an imaginary Stack Computer (the "SC" as described by Wirth) and was initially implemented on a CDC 6000. P4 is the latest of a series of P compilers and supersedes PL/0 and Pascal-S.

The P4 compiler is a pre-compiled executable for Windows XP command line use provided by Scott A Moore. Source code of the compiler (itself written in Pascal) is available for download.

The PrettyP4 linker takes the P4 p-code and translates it into an equivalent P4 bytecode to be interpreted while performing a considerable amount of compression on the original P4 p-code which (for the Propeller Chip) is notionally designed for a completely 32-bit architecture. Compression primarily reduces the size of data in the constant pool (strings and runtime bounds checking information) but also compresses a number of P4 p-codes to allow for efficient storing of various operand sizes, bytes, words or long. The rather reader-unfriendly format of the P4 p-code file (file.p4) is converted to a much more reader-friendly form (file.p4a and file.p4b) and a complete listing of the linked and assembled P4 bytcode is produced (file.lst).

PrettyP4 is P4 Virtual Machine aware so can also reduce the number of P4 opcodes required where the Virtual Machine code is the same regardless of data type. This reduces the number of opcodes and speeds up Virtual machine execution.

The PrettyP4 linker is provided by AiChip Industries and runs as a command line executable. The PrettyP4 linker is currently a 16-bit MS-DOS program and thus only supports 8-dot-3 filenames.

Once the P4 compiler and PrettyP4 linker have executed, two files (Spin-P4.bin and Spin-P4.spin) will have been created which the AiChip_P4_XXX.spin includes with itself. When the AiChip_P4_XXX. spin program is compiled and executed the P4 bytcode will be interpreted and the Pascal program will be executed.

Development Lifecycle

1) Create the pascal source code - file.pas
2) Compile using pcom.exe - produces file.p4 and file.err
3) Run PrettyP4 on file.err - reports any compilation errors
4) Run PrettyP4 on file.p4 - produces a more readable file.p4a
5) Run PrettyP4 on file.p4a - reformats the source for assembly as file.p4b
6) Run PrettyP4 on file.p4b - produces an assembly listing as file.lst
7) Run PrettyP4 on file.lst - produces the executable bytecode as file.bin
8) Load AiChip_P4_XXX.spin and execute

Language Support

The P4 Pascal compiler is a complete compiler in its own right, prmimarily designed to be able to cmpile itself it was never intended to support the full features of standard Pascal but does offer a considerably close approach to that.

The most significant absence is any 'string' type. Strings can only be implemented as arrays of characters.

Other omissions of an unmodified P4 Pascal from Standard Pascal include -

No Procedures or functions as parameters.
No Inter-procedure goto.
Only files of type "text" can be used.
Only predefined (input, output, prr and prd) files can be used.
Mark and Release replaces Dispose.
Curly bracket comments { } are not implemented.
The predeclared identifiers maxint, text, round, page, dispose are not implemented.
The Reset, Rewrite, Pack and Unpack are not implemented.
No undiscriminated variant records.
No output of boolean types.
No output of reals in "fixed" format.
No Set constructors using subranges ('0'..'9').

Primitive Types

The following types are supported -

Boolean - 1-bit (value 0 or 1)
Char - 8-bit (value 0 to 255)
Integer - 16-bit or 32-bit
Real - 16-bit or 32-bit (not implemented yet)

Set - 16-bit or 32-bit (16 or 32 set members)

The P4 Virtual Machine will run with either a 16-bit or 32-bit stack which is selected at runtime. The same code will run regardless of which size is selected and out of range values will be trapped when using 16-bit and 32-bits is required. If the Pascal program does never exceeds the 16-bit limitation it will run equally well on a 16-bit platform as it will on 32-bit. Using real numbers on a 16-bit machine will howver cause considerable errors of numeric resoluton.

Input and Output

Four I/O streams are provided for -

Input - PS/2 keyboard
Output - TV Display
Prd - "Propeller Debugging", serial out on TX, P30
Prr - "Propeller Receive", serial in on RX, P31

These can be defined in the program heading of the Pascal source code -

```
program mypascalfile(input,output,prr,prd);
var ch : char;
begin
  writeln(output,'Hello TV');
  writeln(prd,'Hello Serial Out');
  read(input,ch);
  writeln(output,'Typed : ', ch );
  read(prr,ch);
  writeln(prd,'Echo : ', ch );
end.
```

The I/O streams can be redefined or redirected by modifying the P4 Virtual Machine.

Future Plans

Because the P4 Compiler, PrettyP4 and Virtual Machine are all available as source code, it is possible to enhance the P4 Pascal language to support Propeller specific extensions.

No enhancements or extensions have been planned so far.

Development Tools

Pascal to P4 P-Code Compiler

This is the command line compiler which takes Pascal source code and emits the P4 p-code. The compiler works under Windows XP but not under Windows 98SE.

Executable and Source download from Scott A Moore's site : here

Linker

Executable and Source are included in the AiChip_P4_XXX.zip download available from the Parallax Forum.

Bytecode Interpreter

Source code is included in the AiChip_P4_XXX.zip download available from the Parallax Forum.

Other Compilers

If the P4 Pascal source code is changed to add more functionality to the compiler, it must itself be re-compiled to produce a newer version of the P4 Compiler executable. This requires a third-party Pascal compiler as the P4 Pascal compiler appears to be unable to compile itself these days and the complexities of self-compiling or self-interpreting the p-code of the compiler is too complex to consider as a practical methodology. Unfortunately it seems there are few Pascal compilers which can compile the P4 Compiler source "as is" and thus some modification to the source will be required to get it working. The IP Pascal compiler from Scott A Moore is claimed to compile the P4 Compiler source "as is" but is a non-free commercial product and the demo version too restricted to do so. Alternative compilers which would be worth considering are -

BACI - Ben-Ari Concurrent Interpreter. A compiler and interpreter written by M. Ben-Ari, based on the original Pascal compiler by Niklaus Wirth. A suite of programming tools orientated around p-code with concurrency and associated abilities added. More information : here

Blaster Master Pascal - A pascal compiler based on the QCC compiler from Quake, so everything should be almost like standard pascal. It's almost a mix of the best of both C and Pascal. More information : here

Borland - Borland Software Corporation have released their Turbo Pascal versions 1.0, 3.02 and 5.5 as "Antique Software" in their Software Museum for free download through their CodeGear Developer Network (no registration required). Download Turbo Pascal, Turbo C and Turbo C++ : here

Free Pascal - Free Pascal (aka FPK Pascal) is a 32 and 64 bit professional Pascal compiler. It is available

for different processors: Intel x86, Amd64/x86_64, PowerPC, PowerPC64, Sparc, ARM. Released under the GNU General Public License. More information : here

GNU Pascal - A native and cross-compiler tool for a variety of platforms released under the GNU General Public License. More information : here

Pascal X - a complete Pascal (development) environment with a very fast compiler/interpreter for Windows 95/NT "console". More Information here

Documentation

Pascal Implementation - by Steven Pemberton and Martin Daniels,

Published by Ellis Horwood, Chichester, UK
ISBN: 0-13-653-0311

Read online at Steven Pemberton's site : here

The Pascal User Manual and Report - by Kathleen Jensen and Niklaus Wirth

Published by Springer Verlag 1974, 1985, 1991
ISBN 0-387-97649-3
ISBN 0-540-97649-3

Details here

Links

Compiler Executable and Source Code download : http://www.moorecad.com/standardpascal/p4.html

P4 Virtual Machine interpreter and PrettyP4 Linker download from the Parallax Forum:
AiChip_P4_XXX.zip

Propeller Tool

Propeller tool (also known as Prop Tool and PropTool) is the IDE including Spin/PASM compiler for the Propeller.
It is available as a free download. The current version is 1.2.7.

Known issues and workarounds

Time out

This should be fixed in version 1.1 of the Propeller Tool

The Problem: Sometimes people complain that they have trouble programming the propeller chip. That programming will fail 4 times out of 5 or more. While the programming dialog is still displaying "Loading RAM" or "Verifying RAM", the propeller will restart, loading up whatever happens to be in the EEPROM at the time. The problem happens more frequently with longer programs, so as a program grows over it's development period, it will appear to be a growing problem.
The Diagnosis: The propeller has a fixed time out. Once programming has begun, it has to be finished in a fixed period of time before the time out resets the chip. If the PC has any delays in execution whilst programming the attempt will fail.
The Work Around: After you hit F10 or F11 to compiler and program the propeller, do not switch to another program until the propeller has successfully been programmed. e.g. No filling in time with a little browsing. The time taken to switch applications can slow Prop Tool and cause the time out.
Close down other programs, especially ones that may be doing processing at the same time as Prop Tool.

No Disk

The Problem: When Prop Tool starts up, it displays an error message with "No Disk " in the title bar.

The Diagnosis: This can happen when you have a flashcard reader connected to your PC without a flash card actually inserted.
The Work Around: You can press the Cancel or Continue buttons, and Prop Tool will launch without further problems.
Place a flash card in the reader and it'll stop doing it.

Release Notes

This is the contents of the ReadMe.txt that ships with Prop Tool

Welcome to the Propeller Tool software for the Propeller microcontroller.
This file contains information about the Propeller Tool not found elsewhere.

WHERE TO FIND INFORMATION
Documentation on this product is contained in the Propeller Manual.
Please visit the Parallax web site periodically to find updated software and documentation.
http://www.parallax.com/propeller

SYSTEM REQUIREMENTS
Windows 2000 or later
The recommended processor for the Operating System
The recommended RAM for the Operating System
40 MB Free Hard Drive Space
24-bit, or better, SVGA video card
1 Available USB port or COM port

INSTALLATION
The Propeller Tool is available as an install file downloadable from the Parallax web site. Simply run the downloaded file and follow the prompts. After
installation, run the Propeller.exe program to run the Propeller Tool software.

WHAT'S NEW

Version 1.2.6
---General---
Changed "Plain" element in Preferences' Syntax Elements list to "Regular." This better describes the element.
Enhanced Preferences to display "Use Default" checkboxes as either "Use Regular" or "Use BLOCK" to indicate which setting will actually be used. Updated
the hint descriptions for these as well.
Updated Help menu to include Enhanced Propeller Help and links to Propeller Datasheet v1.2, Propeller Education Kit Labs (pdf) v1.1, Object Exchange
website, and PE Kit Tools and Applications forum thread.
Included Parallax Serial Terminal with installer.

---Bug Fixes---
Fixed bug in Preferences causing the Background option to be available in non-Block elements.
Fixed bug causing Restore button to not update syntax highlighting when the scheme changed as a result of it.

--LIBRARY---
Added 4x4 Keypad Reader v1.0.
Updated FullDuplexSerial to v1.2.
Updated HM55B Compass Module Asm to v1.2.
Updated Memsic2125 to v1.1.
Updated Numbers to v1.1.
Added Parallax Serial Terminal v1.0.
Added PropellerRTC_Emulator v1.0.

Updated Servo32 to v1.5.
Updated Simple_Serial v1.3.
Added SPI_Asm.spin v1.2.
Added SPI_Spin.spin v1.0.

---LIBRARY DEMOS---
Added 4x4 keypad Reader Demo v1.0.
Added HM55B Compass Calibration v1.0.
Deleted HM55B Compass Module.
Added HM55B Compass Module_Serial Demo v1.1.
Added HM55B Compass Module_TVDemo v1.4.
Updated memsic_demo.spin to v1.1.
Added Parallax Serial Terminal Demo v1.0.
Added Parallax Serial Terminal QuickStart v1.0.
Added PropellerRTC_Emulator_Demo v1.0.
Updated Servo32 Demo to v1.5.
Added SPI Asm Demo v1.0.
Added SPI Spin Demo v1.0.

---Misc---
Updated FTDI VCP Driver (USB to Serial) to v2.04.16.
Updated Propeller Datasheet (pdf) to v1.2.
Updated Propeller Manual (pdf) to v1.1.
Updated Propeller Quick Reference (pdf) to v1.6.
Enhanced Propeller Help examples folder structure (formerly Manual examples).
Added PE Kit Labs examples.

Version 1.2.5
---General---
Modified the Block Group Indicators preference to be True by default.

---Bug Fixes---
Enhanced to prevent system-level dialog indicating "No Disk in Drive..." when a drive and/or path is scanned on a removable media drive that has no media in it. This
would occur on some systems with media card readers either upon Propeller Tool startup, during the session, or both.

Version 1.2
---General---
Enhanced circular reference error message to diagram the relationship between objects to make it more clear where the problem is.
Enhanced serial routines to support FTDI VCP Driver v2.4.6 to avoid a possible "Write Error on COMx" message.

Enhanced to automatically check file associations during the first run.

---Bug Fixes---
Fixed bug causing confusing circular reference message when a child references a parent object with multiple instances.
Fixed bug that allowed for the possibility of an invalid circular reference error if two same-named objects existed in two different folders and both appeared along
a branch of the project hierarchy.
Fixed bug in serial routines that caused a "Propeller Not Found..." error message to be unclear when the Serial Search Method is set to a specific port.
Fixed bug in serial routines causing non-existent COM ports to be displayed in error message as COM65535.

--LIBRARY---
Updated H48C Tri-Axis Accelerometer.spin
Updated Servo32v3.spin

---LIBRARY DEMOS---
H48C Tri-Axis Accelerometer DEMO.spin

---Misc---
Updated FTDI VCP (Virtual Com Port) Driver to v2.4.6.

Version 1.1
---General---
Rewrote all serial routines and related items to increase reliability of Propeller chip identification and download process on machines who's CPU and/or other hardware
is heavily burdened. This should significantly decrease the occurrence of "Propeller chip lost on COMx" error messages during download.
Enhanced to prevent software lock-up when accessing serial port hardware that is malfunctioning, misconfigured, or otherwise unusable by the Propeller Tool.

---Bug Fixes---
Fixed bug causing Progress Form to disappear behind the Info Form if focus changed to Info Form.
Fixed bug causing Progress Form to remain visible and "stuck" if communication completed while application is minimized.

---LIBRARY---
Updated AD8803.spin
Removed ADC.spin
Updated Clock.spin
Updated CoilRead.spin
Updated CTR.spin
Updated Debug_Lcd.spin
Removed DS1620.spin

Added Float32.spin
Added Float32A.spin
Added Float32Full.spin
Updated FloatMath.spin
Updated FloatString.spin
Updated FullDuplexSerial.spin
Updated Graphics.spin
Updated H48C Tri-Axis Accelerometer.spin
Updated HM55B Compass Module Asm.spin
Updated Inductor.spin
Updated Keyboard.spin
Added License.spin
Updated MCP3208.spin
Updated memsic2125.spin
Updated Monitor.spin
Updated Mouse.spin
Updated MXD2125 Simple.spin
Added MXD2125.spin
Updated Numbers.spin
Updated Ping.spin
Updated PropellerLoader.spin
Updated Quadrature Encoder.spin
Added RCTIME.spin
Updated RealRandom.spin
Updated Servo32v3.spin
Added Serial_LCD.spin
Updated Simple_Numbers.spin
Updated Simple_Serial.spin
Updated Simple_Debug.spin
Updated Stack Length.spin
Updated StereoSpatializer.spin
Updated Synth.spin
Updated TSL230.spin
Updated TV.spin
Updated TV_Terminal.spin
Updated TV_Text.spin
Updated VGA.spin
Updated VGA_1280x1024_Tile_Driver_With_Cursor.spin
Updated VGA_1600x1200_Tile_Driver_With_Cursor.spin
Updated VGA_512x384_Bitmap.spin
Updated VGA_HiRes_Text.spin
Updated VGA_Text.spin
Updated VocalTract.spin

---LIBRARY DEMOS---
Updated AD8803_Demo.spin

Updated Coil_Demo.spin
Updated Debug_Lcd_Test.spin
Updated Dither.spin
Removed DS1620-Thermometer-v1.0.spin
Updated Float_Demo.spin
Updated FrequencySynth.spin
Updated Graphics_Demo.spin
Updated Graphics_Palette.spin
Updated H48C Tri-Axis Accelerometer Demo.spin
Updated HM55B Compass Module.spin
Updated Inductor Demo.spin
Updated Keyboard_Demo.spin
Updated Memsic_Demo.spin
Updated Microphone_to_Headphones.spin
Updated Microphone_to_VGA.spin
Updated Monitor_Demo.spin
Added MXD2125 Demo.spin
Updated MXD2125 Simple Demo.spin
Updated Ping_Demo.spin
Added Propeller Floating Point.pdf
Added RCTIME_background_Demo.spin
Added RCTIME_foreground_Demo.spin
Updated ReadRandom_Demo.spin
Updated Servo32v3_Demo.spin
Updated SingingDemo.spin
Updated SingingDemoSeven.spin
Updated SpatialSoundDemo.spin
Updated Stack Length Demo.spin
Updated TSL230 Demo.spin
Updated TSL230 Simple Demo.spin
Updated TV_Terminal_Demo.spin
Updated TV_Text_Demo.spin
Updated VGA_512x384_Bitmap_Demo.spin
Updated VGA_Demo.spin
Updated VGA_HiRes_Text_Demo.spin
Updated VGA_Text_Demo.spin
Updated VGA_Tile_Driver_Demo2.spin
Updated VGA_Tile_Driver_Demo3.spin
Updated VocalTractDemo_Mama.spin
Updated VocalTractDemo_Mixer.spin

Version 1.06
---General---
Enhanced serial port configuration options to allow user to include/exclude ports based on port ID or port description. Also, user can specify the search order of ports.

See Edit -> Preferences -> Operation -> Edit Ports for options.
Added Serial Port Search field to Preferences' Operation tab that allows selection of: 1) AUTO (to scan all ports according to serial search preferences), or 2) a
specific port.
Enhanced to be aware of serial port add/remove events the moment they occur.
Enhanced all serial-related error messages to indicate port events and status.
Added support for Auto Recovery of fatal Serial Port Scanning failures.
Updated Object View and Info View to use enhanced hint window code.
Updated Propeller Quick Reference to v1.5.
Updated FTDI USB Virtual COM Port Drivers to v2.02.04.

---LIBRARY---
Added ADC.spin
Added CoilRead.spin
Updated FloatString.spin
Added Inductor.spin
Added MXD2125 Simple.spin
Added Servo32v3.spin
Added Synth.spin
Added TSL230.spin
Added VGA_1280x1024_Tile_Driver_With_Cursor.spin
Added VGA_1600x1200_Tile_Driver_With_Cursor.spin
Updated VGA_HiRes_Text.spin

---LIBRARY DEMOS---
Updated AD8803_Demo.spin
Added FrequencySynth.spin
Added Inductor Demo.spin
Added Memsic_Demo.spin
Added Microphone_to_Headphones.spin
Added Microphone_to-VGA.spin
Added MXD2135 Simple Demo.spin
Updated Ping_Demo.spin
Added Servo32v3_Demo.spin
Added TSL230 Demo.spin
Added TSL230 Simple Demo.spin
Added VGA_HiRes_Text_Demo.spin
Added VGA_Tile_Driver_Demo2.spin
Added VGA_Tile_Driver_Demo3.spin

Version 1.05.8
---Bug Fixes---
Updated compiler to fix bug causing local labels of exactly 16 characters to be processed incorrectly. This was fixed in version 1.05.5 but was

mistakenly broken again in v1.05.6 and v1.05.7.

Version 1.05.7
---Bug Fixes---
Fixed scaling issues with Progress window, Object Info window, and Preferences window, that occur when system has a DPI setting other than 96 dpi.
Fixed to disallow filenames without the proper extension. This is to support .spin, .eeprom, and .binary in one deterministic fashion.
Fixed bug preventing .binary or .eeprom files (listed on the command line) from opening upon initial startup.

Version 1.05.6
Updated compiler to support $ as a "here" operator.

Version 1.05.5
---General---
Added preference item to the Operations tab to control how the Propeller Reset Signal is output. The signal can now appear on the: DTR pin (default), RTS
pin, or on both DTR and RTS pins.
Added "undo after save" preference item to the Files and Folders tab.

---Bug Fixes---
Updated compiler to fix bug causing local labels of exactly 16 characters to be processed incorrectly.
Updated serial communication routines to including scanning of COM ports that don't register normally with the system. This issue was preventing some
manufacturer's COM port devices from being recognized by the Propeller Tool.

Version 1.05.2
---General---
Enhanced to allow stub-loader configurations of binary and eeprom files. Adjusted Info View to display memory info and map in dark gray for everything
that could be code space (based on image size) and medium-gray for everything that is outside that region.
Adjusted look of block syntax preference items.
Removed *.binary and *.eeprom from normal Save As dialog.
Enhanced compiler to:
1) Support a new directive, ORGX, to allow user to stop COG address incrementing for large-model assembly programs.
2) Enhance arguments of ORG, RES, FIT, and ‘repeat’(in BYTE/WORD/LONG value[repeat]) so that they are allowed the same scope as instruction operands.
3) Support RES as _RET destinations.
4) Support TESTN instruction (which is an ANDN instruction, no result write... similar to how TEST is

really an AND, no result write).
Updated syntax highlighting to include ORGX.
Updated syntax highlighting for TESTN.
Enhanced to refresh the file list if a Top Object File was SaveAs'd.
Added help menu items for the Propeller Quick Reference, Propeller Manual and Propeller Demo Board Schematic.

---Bug Fixes---
Fixed bad syntax highlighting when an equal immediately follows a comment in CON section ('=).
Fixed corrupt label on About window that caused immediate exceptions upon execution.
Fixed Progress display to show current compiled code rather than the last object name in the immediate chain.
Fixed bug causing multiple versions to mistake each other's auto-recover files as their own.
Fixed bug causing exceptions upon resizing edits.
Fixed bug causing tab to be activated without updating status bar after a tab was deleted.
Fixed Info Window to properly color code the Info Box and the Memory Map.
Fixed source location methods to prevent rare error when creating an Archive.

Version 1.0
---General---
Added Preferences feature (Edit -> Preferences). Includes options for changing syntax highlighting, file association checks, launching into single
or multiple editors, showing/hiding bookmarks, line numbers, and block group indicators, auto-recover, saving and loading syntax schemes.
Updated color scheme.
Standardized sounds for all messages.
Enhanced forms to reposition themselves if they are more than 50% outside of visible space.
Optimized compilation process.
Enhanced Find/Replace to allow blank Replace fields (so user can replace text with nothing if desired).
Enhanced serial routines to prevent tool from hogging CPU cycles unnecessarily.
Added Auto-Recovery feature; if a system failure occurs, the next session recovers the last-used files up to the point they were last compiled and
relays options to user.
Eliminated limit of 32 objects per Propeller Application.
Modified to decrease startup time by retaining Show Recent Only button state between sessions (press Show Recent Only button to limit Integrated
Explorer's workload).
Enhanced Archive error message "Object View Empty..." to be more clear.

---Bug Fixes---
Eliminated memory leak in compilation process.
Fixed bug preventing undo/redo after saves.
Fixed issue causing the tool to process many key presses twice.
Fixed bug that allowed out-of-range font size values.
Fixed menus from responding during application initialization.

Fixed syntax highlighting of code and doc comments after a LONG declaration in DAT block.
Fixed syntax highlighting of WORD and LONG declarations in DAT block after line end.
Fixed syntax highlighting of assembly local labels after instruction.
Fixed syntax highlighting of = operator after 2 or more spaces in CON block.

VERSION 0.98
---General---
Updated/Added Objects in Library.
Updated compiler to support multi-pass CON/VAR/OBJ blocks to allow CON-defined constants to be used throughout those blocks.
Updated syntax highlighting rules to support IFNOT and ELSEIFNOT reserved words.
Enhanced Info's OpenFile method to indicate that file may not be a Propeller Application file, upon error.
Added Close All Others option to both Edit shortcuts and Edit Tab shortcuts.
Updated compiler to support TRUNC, ROUND, and FLOAT as automatic CONSTANT directives in addition to their normal tasks.
Adjusted parser rules to not use _, $, or % as delimiters so that labels with underscores, or hex or binary numbers are selected properly with double-clicks.
Enhanced serial communication to prevent sticking on invalid ports and to user better feedback.
Modified/Updated Shortcut keys to following:
Ctrl + Shift + B : Show/Hide Bookmarks
Ctrl + Shift + N : Show/Hide Line Numbers
Ctrl + B : Toggle current line's bookmark on/off
Ctrl + N : New file
Ctrl + W : Close current file
ALT + T : Set Current File as Top File.
Removed Minimize/Maximize buttons from Object Info window.
Removed "+ Run" from Load RAM and Load EEPROM menu options. Removed Load EEPROM menu option.
Updated hints. Made corresponding changes to Object Info window. Removed F12 and Ctrl + F12 as a shortcut keys.
Enhanced to allow opening *.binary and *.eeprom files into Object Info window.
Updated compiler to error out when literals greater than 9 bits are used in the source field of assembly instructions.
Enhanced Archive to enabled status all the time; it now prompts user if the Object View is empty.
Updated compile to fix STRING bug when in CASE-OTHER block.

---Bug Fixes---
Fixed bug in Archive feature that caused it to truncate binary files.
Fixed bug causing Edit Tab shortcuts Close and Close All to not necessarily match up with that of Edit shortcuts.

VERSION 0.95.1
---General---

Added copyright notice to About window.
Updated/Added Objects in Library.

---Bug Fixes---
Updated compiler to fix REBOOT command.
Updated Parallax font to v0.70.

VERSION 0.95
Initial pre-release.

This page covers the proposed new Instructions for the Next Propeller chip.

The new chip will have all the the existing instructions with the same binary layout. New instructions have been mapped into the unused slots. (Reference = Chip Gracey Post)

Instructions:

- MUL - multiply instruction. This is the only post I could find with any mention of the multiply.
- Pre/post increment/decrement RDxxxx/WRxxxx enhancement. New SETPTRA/B instructions and PTRA/B registers. See this post for details.
- REP[x,y] - repeat y instructions x times. The repeated instructions will be 2 instructions after this one due to pipelining. See this post for details.
- SWAPZC D - swap the Z and C flags with the 0 and 1 bits in D. See this post for details. (this may be out based on some later discussion)
- JMPD/JMPRETD - delayed jump. The jump would occur two instructions later. This allows the two instructions already in the pipeline to finish and the new instruction after the jump to pipeline in without a stall. See this post for details.
- RDQUADL/WRQUADL - read/write 4 longs to/from HUB memory. See this post for details (also just above it where he first mentions 8 longs but changes to 4).
- ??? - Instructions to read/write a 256 word (16bit) color look up table (CLUT). This post mentions it. This post says that this memory will also be usable as 128 longs of just general data storage.

Other related info:
In this post, Chip mentions a divider circuit. The concept is that you would write your values to registers and then some number of clocks later read the results back out. This same concept is mentioned for square root elsewhere in the same thread.

This post mentions: CORDIC, MAC/MACS, REPeat, indirect register addressing, and hub memory pointers.
Chip mentions in other posts some instructions called SETINDA/B, and associated INDA/B registers. These are the "indirect register addressing" feature. This post has some more info on these.
He, also, mentions the PTRA/B and SETPTRA/B instructions when talking about the post/pre increment/decrement feature enhancement to RDxxxx/WRxxxx. These are the "hub memory pointers" feature.
It's unknown at this time if the PTRA/B and INDA/B registers are just more special registers at the end of cog memory (most likely) or if they are something new.

Propeller Demo Board

I'm going to assume you have the entire Propeller Starter Kit which includes the propeller demo board. (It's possible to get the Propeller Demo Board by itself).

Quick Start installation (is there a better step-by-step tutorial anywhere else?):

- Put the CD in your computer. (It's vaguely titled "Software, documentation, product briefs, catalogs, and image files.")
- choose "Software".
- Open the "Propeller" folder, and choose "Propeller Tool". Hit the "Install" button.
- Hit "next" a bunch of times to install.
- A "Propeller Tool" icon should show up on your desktop.
- *... (There's no need to Hit the "back" button, and choose "Documentation", because the above installation already installed the "Propeller Manual" and the "Propeller Demo Board Schematic".)*
- *... Is there something else I'm supposed to install from the CD? ....*
- *... Are there any updates/patches I'm supposed to download from the web site? ...*
- Now click on the "Propeller Tool" icon, and choose "Yes".
- Connect the Propeller Demo Board to the "wall wart" power supply that came in the Started Kit and turn the power on. (The LEDs should blink in an interesting pattern).
- You can now connect your VGA monitor, TV and speakers to see and hear the demo program in the EEPROM.
- Connect the Propeller Demo Board to your PC using the USB cable that came in the Starter Kit.
- Start the Propeller Tool software on the PC, and press F7 (or select from the top menu "Run" | "Identify Hardware". You should see a "Information" dialog box pop up with a "Propeller ... found" message.

Now you are all set up for

- Propeller Programming Tutorials
- more tutorials.
- yet more tutorials

The kit includes the Propeller Manual. You probably also want to download the Supplement and Errata for Propeller Manual.

- English language Propeller chip forum
- English language HYDRA Game Development Kit forum
- Sprechen Sie Deutsch?
    - Das deutsche Forum zum Parallax Propeller existiert leider schon eine Weile nicht mehr.
    - Zum auf 3 Propeller-Chips basierenden Hive-Computer-Projekt gibt es eine Webpräsenz

mit Forum und dort dreht sich die Welt auch mehr um den Hub als um den Hive, so hivespezifisch wie es der Name suggeriert geht es dort nicht zu. Jeder Propeller-Interessierte ist willkommen.
An english sub-forum exists aswell.

- Habla Espanol? Esta lista de discusión está orientada a usuarios de productos Parallax.

The PropellerTool includes a built in Parallax font that is used to display the special drawing characters, and corresponds to the internal font table in the Propeller ROM. When PropellerTool is started, it will install the font Parallax.ttf on startup.

It seems the current version (0.700) of this font is very sensitive to resolution variations. Often can be seen as bad kerning, in particular on linux all the characters often get totally corrupted both under wine and in native applications. This can be seen by changing point sizes and italic/bold modes.

The easiest fix is to download the a tweaked font file from http://forums.parallax.com/attachment.php?attachmentid=53903&d=1212437073, and replace the existing font.

Linux

In linux you replace the font in two places, wine and Xwindows.
Wine will keep it in ~/.wine/drive_c/windows/fonts/
Gnome/Xwindows location used to be able to use the Gnome file manager to access 'fonts:///', but that no longer works for me (Ubuntu 8.1).

What worked for me was:

```
cd /tmp
wget --content-disposition http://forums.parallax.com/attachment.php?a
ttachmentid=53903&d=1212437073
sudo mkdir /usr/share/fonts/truetype/myfonts/
cd /usr/share/fonts/truetype/myfonts/
sudo unzip /tmp/Parallax.ttf.zip
sudo fc-cache -f -v .
```

Windows

Open the control panel and delete the existing font, and drag in the new version.

From Scratch

The fixed font has all the bitmapped fonts stripped from the Parallax.ttf font file. The resulting vector font then seems to work fine.

The instructions are:

1. copy/backup the original Parallax.ttf file that get puts in the windows font directory.
2. open the Parallax.ttf file with fontforge.
3. When prompted import none of the bitmap fonts.
4. Select Element->Font Info
5. Update the version name and comments. (ONLY!)
6. Choose File->Generate Fonts
7. Overwrite your Parallax.ttf

Note that it seems the font name is embedded in the TTF file, so you cannot simply create a new file name and then be able to use both the old and new font. If you want to do that, go back and edit the font name fields in the Font Info and generate to a new file name.

Mac OSX

On Mac OSX the application **FontBook** can be used to install fonts for the different users. From the menu **File->Add Font** select the directory containing the Parallax.ttf (or any font file for that matter) and press the **OK** button. The file will be installed into ~/Library/Fonts, i.e. the fonts directory for your user. You can also move it there manually. If you do not want the font anymore, just move the file to the trash. (The font Monaco works very well in **BST** too). A font point of 16 with anti-alias (select it from BST editor preferences pane) works very well.

Discussion posted on http://forums.parallax.com/showthread.php?t=98307 and a copy of the modified font file http://forums.parallax.com/attachment.php?attachmentid=53903&d=1212437073

Propeller 2

The next generation of the Propeller chip is currently under development by Parallax. It will undoubtedly be called something featuring the word "Propeller" at release. For now we refer to it as the Propeller 2. Nothing is certain about this chip so far, but Parallax has told the community about a lot of features they are expecting to include.

Early on they had talked about going with 16 COGs, but due to silicon size and other considerations they have gone back to 8. This may change, or they might do another version later with 16.

The primary values that seem to be settled on for now are 8 COGs, 128K Hub RAM (tentatively Beau Schwabe has said 256K is still a possibility), 32K ROM, 92 I/Os, and 160Mhz.

Here is a page with information about proposed: Propeller 2 Instructions.

Here is a link to the Propeller 2 feature list released by Parallax: Propeller 2 Feature List

Features mentioned in posts on the parallax forums:

Feature	Quantity	Comments	Forum post link
Cogs	8 v	COG instructions pipelined (1 per clock effective) Hub instructions take 2 clocks, you can fit 6 regular instructions between successive hub accesses. Quad-long read (four longs in one hub instruction) is on the slate for implementation as well	8 COGs post (other mentions in the newer posts from Aug 2008 to Sept 2008) Other data: Chip post
		HUB access every 8 clocks	Chip post
		512 longs per COG (same number, but more will be available for coding than the Prop 1, 506 vs 496)	Chip post
		256 entry 16bit CLUT, used with VSU to get 16bit color data per pixel.	Chip post Chip post

		Also accessible as 128 longs of general storage. "The CLUT will not be rewritten during a cog reload, so it will retain its prior contents."	
Memory	RAM 128K v	HUB memory Layout Engineer Beau Schwabe assures room on the die for at 128K, and some rearranging may allow for 256K.	Beau Post
	ROM 32K v	ROM includes entire development system. (No need for PC!) Released info from Parallax says 32K.	128k ROM: Chip post Other: Chip post
HUB Address Space	32bit v	In order to have 128KB RAM and 32KB ROM, they had to expand. They decided to go all the way to 32bit.	Chip post
I/Os	92 v	Most recent posts say it's 92 now. "each pair of adjacent Prop II I/Os has a high-speed comparator between them that can toggle at 50MHz"	Chip post Chip post Chip post
ADC/DAC	92 v	"EVERY pin will have one these babies in it, along with a comparator, a delta-sigma ADC, a delta-sigma DAC, a high speed signal/video 75-ohm DAC, pull-ups/downs, slew control, float/weak/strong HIGH/LOW combos, schmitt input w/feedback,	Chip post

		crystal oscillator, and a few other things"	
Serializer / Deserializer	?	"I just need to narrow down what kinds of demodulation we should support. Manchester and NRZ come to mind"	Chip post
PLL speed	160Mhz ?	They hope to reach 160Mhz.	Chip post
Packaging		TQFP-128 (14x14mm) ? QFN-128 (12x12mm) ?	Chip post Chip post
Process	180nm v		Beau post
Pins	128	92 I/Os v 8 VP0-7 power 1 per 8 I/Os (1.8v - 3.3v) (more functional at 3.3v) v 8 GP0-7 grounds 1 per 8 I/Os v 8 VDD 1.8v Core power v 8 GND v 1 RESn Reset v 2 XI/XO Clock v 1 BOEn Brown Out v	VIO pins: Chip post Pin arrangement Image from Beau: Image

legend: ? = not yet defined by Parallax | v = mentioned by Parallax (Chip, Paul, Beau)

The information on this page and the Propeller 2 Instructions page was gathered primarily from the following posts:

- What would you want more of, cogs or RAM? (started 24Nov06)
- More Prop II info..!?! (started 21Aug08)
- Should the next Propeller be code-compatible? (started 27Aug08)
- Prop II on-chip development question (started 22Nov09)

Propeller Lingo

This page is dedicated to the growing number of Propeller specific terms, used elsewhere in this WikiSpace. Unlike most all other CPUs, the Propeller is a multi-processor design, featuring deterministic timing, no interrupts and highly flexible on board hardware and I/O capability.

What is a COG?

A cog refers to one of 8 internal CPU cores. Each COG runs independent of the others, has a 2KB memory space, apart from the shared 32KB HUB memory space. Think of a COG as one element of a multi-processor system. All cogs share access to the I/O pins in real time. It is possible to have one COG watching and acting on what another COG does with the I/O pins.
When not active, COGs do not consume power.
When active, a COGs memory space is filled from the HUB, then program execution begins.
All COGs are identical, featuring on board counters, video generator, etc... Programs have no need to specify specifc COGs to execute on, unless some hard coded software need warrants this.

What is the HUB?

On the Propeller, there is a shared memory space called the HUB. The 16 bits of addressable space is evenly divided into RAM and ROM areas, both accessible to programs running on a COG. Each COG gets round robin access to the shared HUB memory space. This was done to make the design completely deterministic, thus reducing the complexity normally associated with multi-processor designs.

CNT --or-- Global System Counter

This is the read only system wide counter. It is visible to all COGs and is 32 bits in size. Typical use case is to store the current count, calculate some offset, then compare it within a COG to establish known and deterministic timing between COGs, or as a base counter for other deterministic tasks within a given COG that may or may not be using it's own counters.

What is SPIN?

Spin is a higher level language, run from an internal and on chip interpeter. Spin programs are compiled with the Propeller tool, and run on one or more COGs.

What is Spin Bytecode?

Spin bytecode is the sequence of bytes which make up the instructions and data of a compiled Spin program. The Spin Bytecode is executed by the Spin Interpreter which is loaded from ROM into any COG which is required to run a Spin program.

What is PASM?

Propeller ASseMbly. Machine code, nirvana, or the cause for delicate nerves.

What is LMM?

Large Memory Model. This is a form of PASM where Propeller assembly instructions are held in Hub memory rather than in COG memory and are interpreted by the COG program rather than directly executed.
There is no single version of LMM but they all have core functionality in common; the ability to execute most non-branching Propeller instructions at high-speed with those instructions which cannot be directly executed replaced by a jump into the LMM handling COG program to perform the task required.
The Propeller and Propeller Tool do not natively support LMM. LMM implementations have to be designed and implemented by Propeller developers themselves.

What is a VM or Virtual Machine?

A Virtual Machine is a Spin, or more usually a COG, program which can interpret a sequence of instructions stored somewhere within or external to the Propeller chip and cause the intended operations of the interpreted code to occur.
A Virtual Machine allows a processor to execute an instruction set which is not native to it, that is, is not in its own native assembly language.
The ROM-based Spin Interpreter and LMM handling COG programs are Virtual Machines.

What is a WAITVID?

"**WAIT** for **VID**eo generator to be ready to accept pixel data." It's an assembly instruction, often used in plain English to describe the task of feeding pixels to the hardware. eg: "My driver delivers 4 pixels per WAITVID."

What is a Register?

On the Propeller, there are two distinct memory spaces, one being HUB memory, the other being COG memory.

In the simplest sense, a "register" on the Propeller really is just one of the 512 COG memory locations. In the not so simple sense...

Many CPU's have any number of internal registers, used for operations on data that resides somewhere in the addressable memory space. (A, X, r1, etc...) These registers are typically addressed in a way distinctly different from ordinary RAM or ROM memory. eg: LDA $100 That particular instruction, for 6809, 6502, and probably others, instructs the CPU to move the contents of memory location $100 into register A, by way of example, where A is an internal register, with no address other than the bit field in the instruction that specifies it, and $100 being one of the addressable memory locations.

The Propeller does not make use of this model, in that all 512 addressable COG memory locations simply contain values, with said values either being instructions or data, based on the programmers intent. On the prop then, a similar instruction would be: MOV A, $100

The difference being A is just a label, pointing to another COG memory location, instead of referring to some internal location! One could just as easily do this: MOV frank, $100. If frank and A both point to COG memory location $1a0, for example, then the contents of location $100, would be moved to location $1a0.

Essentially, if you plan on executing the contents of a given COG memory location, it then works like an instruction. If those contents are to be consumed or operated on by the program, then it's more like data. The most common use of the word "register", in the context of the Propeller, is to refer to a COG memory location that contains data, to be operated on according to instructions given, in a fashion similar to how an internal register would be used on other CPU models.

What is PropJavelin?

PropJavelin is a project to implement the functionality of the Parallax Javelin Stamp on the Propeller Chip. This is the implementation of a JVM which runs on the Propeller to allow Java(TM) programming of the Propeller. Java program development is undertaken using a modified version of the Javelin Stamp IDE.

More information on PropJavelin and programming the Propeller with Java can be found here

Propeller Manual

by Jeff Martin

ISBN 1-928982-38-7

A 438 page book which comes included in the box with the Propeller Starter Kit.

Also online.

errata:

- http://www.parallax.com/dl/docs/prod/prop/PMv1.0Supplement-v1.1.pdf
- Antilog tables: There is an error in the propeller manual 1.0 describing how to use the antilog (2^x) tables. a discussion thread tells the right way to do it. It should read: `table_antilog long $D000 'anti-log table base`
- minor bug in the loadTable routine used by the log and exp functions has been fixed.

Propeller Tool Enhancement Requests

This is a list of enhancements which Propeller Developers believe could be made to the Propeller Tool to provide more functionality than the latest released version delivers. It is also a useful guide to what a third-party developer of programming tools for the Propeller may wish to consider implementing to satisfy the perceived needs of developers.

This page has been split into three sections -

New Feature Requests - Features which can be implemented in an IDE or other front-end to enhance the compilation process or programming languages (Spin and/opr PASM) which can be achieved through pre-processing in the IDE or using another application and do not require a change in the compilation process.

Compiler Enhancement Requests - Enhancements which can only be provided for by a change to the compiler itself and cannot be achieved through pre-processing by the IDE or other application before compilation.

Propeller Tool Improvement Requests - Enhancements which are suggested for the Parallex Propeller Tool IDE.

New Feature Requests

Macros and Conditional Compilation

C-style #include, #define, #if-else-endif and similar pre-processor directives.

Third-Party Hooks / Plug-Ins

1) A mechanism to integrate third-party pre-processing tools into the compilation process.

2) A mechanism to allow third-party downloaders to be used instead of the in-built IDE downloader. This will allow end-users to use three-wire (RX/TX/0V) download mechanisms which do not have DTR signalling for Propeller Reset.

3) A mechanism to launch third-party tools after download completes, for example third-part debugging tools and and terminal emulators.

Separate Tokeniser

A separation of the IDE front-end and Spin Compiler back-end would be desirable for those wishing to write their own front-ends, IDE's and GUI's.

Ideally multi-platform command-line tokenisers would be made available.

Library Paths

The means to supply user-specified paths which can be used to locate pre-written Spin modules which have been specified in a program's OBJ section.

Paths Within Object Specifications

The current definition of a sub-object in the OBJ section of a program is name : "filename". The filename should allow a path to specified, an absolute path, or relative to the directory the current object (that including the sub-object) has been saved in.

Object Base Address Identifier

A mechanism like "@Mymethod" which allows the start of an object to be identified within hub memory.

Method Index Identifier

A mechanism to determine the index of a method (subroutine or function) within the vector table held within an object through which a method call is made.

String Packing

Compile-time directives additional to String() to allow 5-bit, 6-bit and 7-bit character encoding and packing within hub memory, with and without terminating zero.

Compile-time directives to allow 5-bit, 6-bit and 7-bit character encoding and packing within a 32-bit long variable, left or right aligned.

String Hashing

A compile-time directive which will return a 32-bit constant being a hash of a supplied string. Ideally this

would use the same syntax as the "String()" directive but return a hash value rather than a pointer to the string.

Pre-Defined String Constants

Compile time constants which work like String() but return pointers to text strings which indicate the top-object filename, current filename, compilation date (including non-US date formats) and time.

SizeOf Operator

A SizeOf() compile-time directive which returns the number of bytes reserved for a variable, or number of bytes reserved for an 'array'.

Address Variable Type

Provide an "Addr" variable type of the correct size to hold a hub address of the Propeller Chip for which the code is being compiled for. This would be a word for the Propeller Mk I, a long for the forthcoming Propeller Mk II.

PASM should also be updated to provide rdaddr and wraddr as equivalents for rdword/rdlong and wrword/wrlong and Spin also updated to include addr[ ] arrays.

Top-Object Only Download

A mechanism to avoid accidentally downloading and running anything other than a top-object. One suggestion is to check for the presence of _CLKMODE in the open file being downloaded. If present allow downloads, if not present allow a syntax check F9, but do not allow an F10/F11 download.

Enhanced Assembly Options

A mechanism to cause the use of long or other PASM instructions used after 'res' to be flagged as an error. The option should be user-selectable so it can be disabled for those cases where such use is intentional.

Target Identification

A means to identify whether source code is targeted at Mk I, Mk II, or both Propeller Chips and a mechanism to inform the Propeller Developer that they are using incompatible source for their intended target.

Automatic generation of .eeprom or .binary files

A mechanism to allow the automatic generation of .eeprom or .binary files with every compilation without having to perform an explicit F8 or confirm the deletion of existing .eeprom or .binary files, while deleting any outdated .eeprom and .binary files.

Compiler Enhancement Requests

Listing and Symbol Table File

The production of human-readable and/or computer-readable listing and/or symbol tables of the compilation to allow the location of labels to be related to hub memory address (and cog address where relevant) for that compilation.

Current Hub Address Identifier

An "@" token which will specify current hub address within DAT sections in a similar way that the "$" token is used to specify current cog address. This would be particularly useful for developers of LMM code and VM's.

Propeller Tool Improvement Requests

Better COM/Serial Port Selection/Polling

This has been implemented by Parallax staff and is available from version 1.06 (2008-01-22) - Thanks Jeff (Parallax).

A Debug Terminal Display

This is currently scheduled for implementation by Parallax staff.

Tabs

1) The ability to reorder (organize) the tabs when several tabs are open.

2) The "x" for closing the tabs to be located on the tab itself instead of right most side of the window. Similar to Internet Explorer 7.

Windows 2000 compatibility

Probably not high on most people's lists

Printing

1) Ability to print in landscape orientation

2) Remember my settings (e.g. 'Header', and 'Line numbers')

Compiler output

Add option to always generate the .BINARY and .EEPROM files (instead of running the internal loader). This would allow the use of external loaders for linux and mac users. It would also allow two propellers (or more) propellers to be attached via USB and programmed by the external loader (for example, you could run two copies of Loader.py in watch mode, watching different binary files and connected to two different USB ports) by just compiling the appropriate top level object.

Configuration Override

The ability to specify a _CLKMODE and _XINFREQ which overrides whatever is specified in source code. Most home users will use one Propeller board or multiple boards configured the same and this will save having to alter other source code to match the target hardware they have. This is particularly useful should the crystal be some unusual frequency prone to typing error. It also allows those users to deliver source code set for the standard XTAL1+PLL16x / 5MHz while developing with whatever they are actually using.

A Propeller+CPLD project by Pacito.Sys

The goal of this project is to use a CPLD to augment the video circuitry of the propeller to obtain high resolution frame-buffer based video.

A CPLD, complex programmable logic device, provides the glue logic needed to access a external SRAM where the frame-buffer is stored and also provides a way to access the SRAM from the propeller using as few lines as possible.

A thread at Parallax' forums can be found here.

The circuit is shown below:

CPLD and SRAM

The circuit uses 2 CPLD, one XC9572 and one XC9536 because is what I had laying around. If you get a XC95144(XL) or bigger you can fit both parts into the same CPLD.
The color capabilities rise to 256 simultaneous colors from a palette of the same amount using 3:3:2 for R:G:B. In a FPGA a better wider DAC could be used and a palette could be held in memory but that is "Zukunftsmusik".

How it works:

Video refresh. The propeller generates synchronism signals for vertical and horizontal as well as a pixel enable and pixel clock signal. This pixel clock signa has to be 2 times the pixel clock. The pixel enable signal has to be asserted during the visible cicle of the image. Every other cycle the memory will be ready to be read or written by the propeller. As the CPLD use is a bit of a tight fit no two consecutive cycles can be used for read or write during blanking but it could be in a bigger device.

The memory waveform explaining this interleaved process is shown below. Here the simulator and test-bench signals are shown. The software used is Xilinx WebISE 10.1 but newer or older versions work as well.

A logic analyzer hooked to the memory and clock signals show the following.

Image here

The actual board has been done as a single side PCB with few bridges. The photo process worked quite well, for being the third board I do using this process. Details can be found here. Note that removing the oil before developing is very important for the NaOH to attack the photo sensible layer!

Image here

Fully built circuit can be seen here

The Code inside the XC9572 is shown below (Verilog), it is fixed for 640x480 i.e. 307200 pixels.

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////
////////////
// Company: Pacito Systems Co.
// Engineer: Pacito.Sys hppacito <@> gmail.com
//
// Create Date:    18:04:13 10/03/2009
// Design Name:
// Module Name:    top
// Project Name:
// Target Devices: XC9572(XL)
// Tool versions: WebISE 10.1 (Linux)
// Description: A VGA and memory controller for the propeller
//
// Dependencies:
//
// Revision:
// Revision 0.02 - The finite state machines are used to handle memory
 read and write
// Additional Comments: (c) Copyright 2009 R.A. Paz Schmidt (aka Pacit
o.Sys, Pacito Systems Co.)
//
//////////////////////////////////////////////////////////////////////////
////////////
module top(
```

```
    output [18:0] m_addr, // Memory address
    output m_nce, // memory chip enable
    output m_noe, // memory output enable
    output m_nwe, // memory write enable
     output [1:0] wstate, // only for simulation
    output i0_read_le, // latch enable when reading
     output i1_vga_le, // vga latch enable
    output i2_mem_be, // buffer enable for memory write cycle
    input in_clk, // input clock, two times desired pixel clock
    input in_pixel_en, // clock enable, pixel clock will be active whe
n this signal is 1
    input in_dataw, // data write signal
    input in_datar, // data read signal
    input [7:0] in_data // data bus from propeller
    );

parameter MEM_AWIDTH = 19;
parameter MEM_DWIDTH = 8;
parameter MAX_PIXELS = 11'b01010110000;
reg [MEM_AWIDTH-1:0] r_pixel_counter;
reg [MEM_AWIDTH-1:0] r_memory_ptr;

reg clk_pixel;
reg [1:0] r_wstate, r_nwstate;
reg [1:0] r_rstate, r_nrstate; // not needed
reg r_pixel_en;

reg [1:0] r_ridx;

reg r_mem_noe;
reg r_mem_nwe;

wire w_video_noe;

wire clk_mem = ~clk_pixel;

wire w_cmd_write = in_datar & in_dataw; // command write when both are
 asserted
wire w_dataw = in_dataw & (in_dataw ^ in_datar); // internal write sig
nal
wire w_datar = in_datar & (in_dataw ^ in_datar); // internal read sign
al

assign wstate = r_wstate;

// every other clock we refresh the display if needed
```

```
always @ (posedge in_clk)
begin
   clk_pixel <= ~clk_pixel;
    if (in_pixel_en == 0) begin
        r_pixel_en <= 0;
    end else begin
        r_pixel_en <= 1;
    end
end

// Command write on the rising edge
always @ (posedge w_cmd_write)
begin
    r_ridx [1:0] <= in_data[1:0];
end
// Value write to the right register on the falling edge
always @ (negedge w_cmd_write)
begin
   case (r_ridx[1:0])
        2'b00: r_memory_ptr[7:0] = in_data;
        2'b01: r_memory_ptr[15:8] = in_data;
        2'b10: r_memory_ptr[MEM_AWIDTH-1:16] = in_data[MEM_AWIDTH-17:0
];
        //2'b11: r_memory_ptr[MEM_AWIDTH-1:16] = in_data[MEM_AWIDTH-17
:0]; // we do the same...
    endcase
end

// Increments pointer
always @ (posedge clk_mem)
begin
    if (r_pixel_counter[18:8] == MAX_PIXELS)
        r_pixel_counter = 0;
   else
        if (r_pixel_en == 1)
        r_pixel_counter = r_pixel_counter + 1;
end

// CPLD <-> Propeller comm
// Write

always @(*)
begin
    r_nwstate = 0;
    r_mem_nwe = (r_wstate == 2) ? 0:1;
    if (r_wstate == 0)
```

```
        if (w_dataw) r_nwstate = 1; // changes state when w_dataw goes
 high
    if (r_wstate == 1)
        if (clk_mem) r_nwstate = 2; // changes state when memory cycle
 is active
        else r_nwstate = 1; // keep current state
    if (r_wstate == 2)
        if (clk_mem) r_nwstate = 2; // keeps current state
end

always @ (posedge in_clk)
    r_wstate <= r_nwstate;

always @(*)
begin
    r_nrstate = 0;
    r_mem_noe = (r_rstate == 2) ? 0:1;
    if (r_rstate == 0)
        if (w_datar) r_nrstate = 1; // changes state when w_dataw goes
 high
    if (r_rstate == 1)
        if (clk_mem) r_nrstate = 2; // changes state when memory cycle
 is active
        else r_nrstate = 1; // keep current state
    if (r_rstate == 2)
        if (clk_mem) r_nrstate = 2; // keeps current state
end

always @ (posedge in_clk)
    r_rstate <= r_nrstate;

// Memory interface
assign w_video_noe = ~(clk_pixel & r_pixel_en);

assign m_addr = (w_video_noe == 0) ? r_pixel_counter:r_memory_ptr;
assign m_nce = r_mem_nwe & r_mem_noe & w_video_noe;
assign m_noe = r_mem_noe & w_video_noe;
assign m_nwe = r_mem_nwe;

assign i0_read_le = r_mem_noe;
assign i1_vga_le = w_video_noe;
assign i2_mem_be = !r_mem_nwe;

initial
begin
    clk_pixel = 0;
```

```
    r_wstate = 0;
    r_nwstate = 0;
    r_pixel_counter = 0;
    //r_mem_noe = 1;
    //r_mem_nwe = 1;
end

endmodule
```

The text code for the propeller is shown below

```
' VGA driver by Chip Gracey (c) 2006 Parallax Inc.
' CPLD code by me (c) 2009 Pacito.Sys

CON

    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000


    VGASYNCPINS = 8
    VGACLK = 27
    VGAPIXEN = 26
    DATAR = 25
    DATAW = 24

  hp = 640      'horizontal pixels
  vp = 240      'vertical pixels
  hf = 24        'horizontal front porch pixels
  hs = 40       'horizontal sync pixels
  hb = 128       'horizontal back porch pixels
  vf = 9        'vertical front porch lines
  vs = 3        'vertical sync lines
  vb = 28       'vertical back porch lines
  hn = 1        'horizontal normal sync state (0|1)
  vn = 1        'vertical normal sync state (0|1)
  pr = 25       'pixel rate in MHz at 80MHz system clock (5MHz granula
rity)

' Tiles

  xtiles = hp / 32
  ytiles = vp / 32
```

```
' H/V inactive states

  hv_inactive = (hn << 1 + vn) * $0101


OBJ
   term : "FullDuplexSerial"


VAR
    byte buffer[16]


PUB start

    cognew(@vgasync, 0) ' starts VGA subsystem
    cognew(@memctrl, @buffer)
    term.start(31, 30, 0, 115200)
    term.str(string("Test, new run...", 13))
    term.hex(buffer[0], 2)
    term.hex(buffer[1], 2)
    term.hex(buffer[2], 2)
    term.hex(buffer[3], 2)
    term.tx(13)


DAT
{ This COG generates the synchronismus signals needed for VGA refresh
  the clock signal and the pixel enable signals
}

                        org     0

memctrl                 mov     DIRA, c6_cnt_dira
                        mov     c6_v_buff, PAR
                        mov     c6_v_addr, #511
                        mov     c6_v_data0, #$00
                        call    #c6_writedata
                        mov     c6_v_data0, #$ff
                        mov     c6_v_addr, #1
                        call    #c6_writedata
                        mov     c6_v_addr, #511
                        mov     c6_v_data0, #$77
                        call    #c6_writedata
                        mov     c6_v_addr, #51
                        mov     c6_v_data0, #$00
```

```
                        call    #c6_writedata

                        mov     c6_v_addr, #511
                        call    #c6_readdata
                        wrbyte  c6_v_data0, c6_v_buff
                        add     c6_v_buff, #1
                        wrbyte  c6_v_data1, c6_v_buff
                        add     c6_v_buff, #1
                        wrbyte  c6_v_data2, c6_v_buff
                        add     c6_v_buff, #1
                        wrbyte  c6_v_data3, c6_v_buff
                        add     c6_v_buff, #1

                        jmp     #$

c6_writeaddr            or      DIRA, #$ff
                        mov     OUTA, c6_cnt_reg0
                        mov     OUTA, c6_v_addr
                        mov     OUTA, c6_cnt_reg1
                        shr     c6_v_addr, #8
                        mov     OUTA, c6_v_addr
                        mov     OUTA, c6_cnt_reg2
                        shr     c6_v_addr, #8
                        mov     OUTA, c6_v_addr
                        andn    DIRA, #$ff
c6_writeaddr_ret        ret


c6_writedata            call    #c6_writeaddr
                        or      DIRA, #$ff
                        mov     OUTA, c6_v_data0
                        or      OUTA, c6_cnt_dataw
                        andn    DIRA, #$ff          ' data has been la
tched


                        andn    OUTA, c6_cnt_dataw
c6_writedata_ret        ret

c6_readdata             call    #c6_writeaddr

                        or      OUTA, c6_cnt_datar
                        andn    OUTA, c6_cnt_datar
                        mov     c6_v_data0, INA
                        mov     c6_v_data1, INA
                        mov     c6_v_data2, INA
```

```
                        mov     c6_v_data3, INA
                        'andn    OUTA, c6_cnt_datar
c6_readdata_ret         ret

c6_cnt_cmdw
c6_cnt_reg0             long    (1<<DATAR)|(1<<DATAW)
c6_cnt_reg1             long    (1<<DATAR)|(1<<DATAW)|1
c6_cnt_reg2             long    (1<<DATAR)|(1<<DATAW)|2

c6_cnt_pe               long    (1<<VGAPIXEN)
c6_cnt_dataw            long    (1<<DATAW)
c6_cnt_datar            long    (1<<DATAR)
c6_cnt_dira             long    (1<<DATAR)|(1<<DATAW)|(1<<VGAPIXEN)
c6_v_addr               long    0
c6_v_buff               long    0
c6_v_data0              long    0
c6_v_data1              long    0
c6_v_data2              long    0
c6_v_data3              long    0

DAT
{ This COG generates the synchronismus signals needed for VGA refresh
  the clock signal and the pixel enable signals
}

                        org     0

vgasync                 mov     DIRA, c7_cnt_dira

                        movi    ctra, #%00001_101        'enable PLL i
n ctra (VCO runs at 4x)
                        movi    frqa, #(pr / 5) << 3     'set pixel ra
te

                        mov     ctrb, #VGACLK          ' clock output
                        movi    ctrb, #%00010_110        ' PLL 2 times
 pixel rate

                        movi    frqb, #(pr / 5) << 3     'set pixel ra
te
                        mov     vcfg,reg_vcfg           'set video con
figuration

' Main loop, display field and do invisible sync lines

field                   mov     color_ptr,color_base    'reset color p
```

```
ointer
                        mov     pixel_ptr,pixel_base    'reset pixel p
ointer
                        mov     y,#ytiles               'set y tiles
:ytile                  mov     yl,#32                  'set y lines p
er tile
:yline                  mov     yx,#2                   'set y expansi
on
:yexpand                mov     x,#xtiles               'set x tiles
                        mov     vscl,vscl_pixel         'set pixel vsc
l

:xtile                  rdword  color,#0                'get color wor
d
                        mov     color,hv                'set h/v inact
ive states
                        rdlong  pixel,#0                'get pixel lon
g
                        waitvid color,#0                'pass colors a
nd pixels to video
                        djnz    x,#:xtile               'another x til
e?

                        mov     x,#1                    'do horizontal
 sync
                        call    #hsync

                        djnz    yx,#:yexpand            'y expand?

                        djnz    yl,#:yline              'another y lin
e in same tile?

                        djnz    y,#:ytile               'another y til
e?

                        'wrlong   colormask,par          'visible done
, write non-0 to sync

                        mov     x,#vf                   'do vertical f
ront porch lines
                        call    #blank
                        mov     x,#vs                   'do vertical s
ync lines
                        call    #vsync
                        mov     x,#vb                   'do vertical b
ack porch lines
```

```
                        call    #vsync

                        jmp     #field                  'field done, l
oop


' Subroutine - do blank lines

vsync                   xor     hvsync,#$101            'flip vertical
 sync bits

blank                   mov     vscl,hvis               'do blank pixe
ls
                        waitvid hvsync,#0
hsync                   mov     vscl,#hf                'do horizontal
 front porch pixels
                        waitvid hvsync,#0
                        mov     vscl,#hs                'do horizontal
 sync pixels
                        waitvid hvsync,#1
                        mov     vscl,#hb                'do horizontal
 back porch pixels
                        waitvid hvsync,#0
                        djnz    x,#blank                'another line?
hsync_ret
blank_ret
vsync_ret               ret


' Data

reg_dira                long    0                       'set at runtim
e
reg_dirb                long    0                       'set at runtim
e
reg_vcfg                long    $200002ff                       'set a
t runtime

color_base              long    0                       'set at runtim
e (2 contiguous longs)
pixel_base              long    0                       'set at runtim
e

vscl_pixel              long    1 << 12 + 32            '1 pixel per c
lock and 32 pixels per set
colormask               long    $FCFC                   'mask to isola
```

```
te R,G,B bits from H,V
hvis                    long    hp                      'visible pixel
s per scan line
hv                      long    hv_inactive             '-H,-V states
hvsync                  long    hv_inactive ^ $200      '+/-H,-V state
s

c7_cnt_dira             long    (1<<VGACLK)|(1<<VGAPIXEN)|(3<<VGASYNCP
INS)

' Uninitialized data

color_ptr               res     1
pixel_ptr               res     1
color                   res     1
pixel                   res     1
x                       res     1
y                       res     1
yl                      res     1
yx                      res     1
```

An image on a TFT monitor at 640x480 can be seen below. Note the use of graphics not possible with a bare propeller.

Propeller Magazine

2008

- June 2008

- May 2008
- April 2008
- March 2008

Editing the Magazine

Anyone can edit the Magazine; simply select the latest issue then click on the **Edit Page** link at the top.

To create a new section use == before and after the section title, for example ...
```
==Advertisements==
```

To add a link, select or create an appropriate section, then on the first line describe what the link is, then on a second line give the link to the Propeller Forum post with [[before and]] after the link, for example ...

```
==Recent News==

Prop Magazine migrates to Propeller Wiki
[[http://forums/p.../forums/default.aspx?f=25&m=263618]]
```

Prop Magazine - March 2008

Next issue April 2008

The Prop Magazine, with many thanks to the work put in by Fred Hawkins.

http://forums.parallax.com/forums/default.aspx?f=25&m=256872

Shock News

DeSilva Tosses Towel (or takes Sabbatical)
http://forums.parallax.com/forums/default.aspx?f=25&m=256944
Sabbitical seems likely, see book below

Basics

bookmarked Propeller manual pdf, w/ google search tips, Ray's logic link
http://forums.parallax.com/forums/default.aspx?f=25&m=254507

external web pages: "Building blocks"
Useful information for new propeller users.
Simple audio interface boards get a mention with two pictures.
http://forums.parallax.com/forums/default.aspx?f=25&m=251102

book: DeSilva outlines his How to start Microcontrolling using the Parallax Propeller,
http://forums.parallax.com/forums/default.aspx?f=25&m=257218

cognew or coginit and kiss thoughts. (IE don't make stuff hard to do)
http://forums.parallax.com/forums/default.aspx?f=25&m=257791

clear thinking is possible -- kiss
http://forums.parallax.com/forums/default.aspx?f=25&m=255146

division, spin,
with tips for flogging integer math to do floating point work
http://forums.parallax.com/forums/default.aspx?f=25&m=258422

spin instruction timing
http://forums.parallax.com/forums/default.aspx?f=25&m=259389

variables between cogs
http://forums.parallax.com/forums/default.aspx?f=25&m=257464

Spin, global variables across cogs and objects: discussion.
http://forums.parallax.com/forums/default.aspx?f=25&m=254609

bit masking, in spin and assembly
http://forums.parallax.com/forums/default.aspx?f=25&m=256775

spin, bit masking. Code fragments included.
http://forums.parallax.com/forums/default.aspx?f=25&m=260279

controlling servos, using Beau's servo32v3.spin (in Prop Tool, starting with version 1.06).
http://forums.parallax.com/forums/default.aspx?f=25&m=255594

lots to chew on -- laptop serial to prop, robot fits where?
http://forums.parallax.com/forums/default.aspx?f=25&m=258177

prop plug and reset avoidance, w/ nested links to previous threads.
http://forums.parallax.com/forums/default.aspx?f=25&m=258255

mosquito's timer module
http://forums.parallax.com/forums/default.aspx?f=25&m=258441

Prop Demo board's tv clipping
http://forums.parallax.com/forums/default.aspx?f=25&m=259567

prop UI theory discussion
http://forums.parallax.com/forums/default.aspx?f=25&m=259991

Basics and Beyond

CommentedFullDuplexSerial.spin (grasshopper's collation of Mike Green's wisdom)
http://forums.parallax.com/forums/default.aspx?f=25&m=258162

Assembly: Sample code that uses a cog's $1f0-1ff special purpose registers as part of the program space.
http://forums.parallax.com/forums/default.aspx?f=25&m=256591

Assembly, ina is source only. Plus nice code minimalization discussion. (w/ examples)
http://forums.parallax.com/forums/default.aspx?f=25&m=258143

assembly, remember sign of cnt when comparing. (w/ code sample.)
http://forums.parallax.com/forums/default.aspx?f=25&m=260776

synchronizing cogs, program counter
http://forums.parallax.com/forums/default.aspx?f=25&m=257095

PropMonitor, PropTerminal links
http://forums.parallax.com/forums/default.aspx?f=25&m=257718

hyperterminal settings
http://forums.parallax.com/forums/default.aspx?f=25&m=259697

Spin, 1 second timers:
http://forums.parallax.com/forums/default.aspx?f=25&m=254589

event timestamping, spin
http://forums.parallax.com/forums/default.aspx?f=25&m=256874

simple ADC, link to rayman's web page with program zip, schematic and resistor calculator applet.
sending serial data (to Pololu Micro Dual Serial Motor Controller)
http://forums.parallax.com/forums/default.aspx?f=25&m=256547

spin and assembly, towards a micro-stepping controller program.
http://forums.parallax.com/forums/default.aspx?f=25&m=261026

more simple ADC in spin.
http://forums.parallax.com/forums/default.aspx?f=25&m=256645

bare bones adc links and Perry's vid capture code
http://forums.parallax.com/forums/default.aspx?f=25&m=260159

spin execution speed, profiling
http://forums.parallax.com/forums/default.aspx?f=25&m=256763

on FloatMath timing (a bit). & Stevenmess' remark on Float pdf error.
http://forums.parallax.com/forums/default.aspx?f=25&m=260417

towards closed loop controller (industrial)
http://forums.parallax.com/forums/default.aspx?f=25&m=257608

Spin syntax, OR's in Cases: Peter's Ladder Logic thread (below) spun out of this thread where Ron suggests the CASE construct is much like PLC's.
http://forums.parallax.com/forums/default.aspx?f=25&m=256295

"Is my propeller dead?", troubleshooting gambits. Not mentioned, low 9v battery on ed kit
http://forums.parallax.com/forums/default.aspx?f=25&m=255083

spin, LOCKSET, LOCKNEW review
http://forums.parallax.com/forums/default.aspx?f=25&m=259235

spin, LOOKDOWN LOOKUP discussion

http://forums.parallax.com/forums/default.aspx?f=25&m=259987

spin, ascii string to integer conversions and, marginally, gps/rtc stuff.
http://forums.parallax.com/forums/default.aspx?f=25&m=258771

spin to assembly, advice
http://forums.parallax.com/forums/default.aspx?f=25&m=259580

spin niceties: i/o pin notation () versus subscript declaration []
http://forums.parallax.com/forums/default.aspx?f=25&m=260027

spin, writing long variables (remember: byte by byte transfer from @address)
http://forums.parallax.com/forums/default.aspx?f=25&m=260064

spin strings, function referencing (misleading metaphor) Links to workarounds
http://forums.parallax.com/forums/default.aspx?f=25&m=260504&p=1

Possible? guitar to midi conversion
http://forums.parallax.com/forums/default.aspx?f=25&m=260907

Sine wave generation
http://forums.parallax.com/forums/default.aspx?f=25&m=261056

PropDOS 1.6 released by Oldbitcollector
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=224206

and then a full text editor for PropDOS -- "Think of this as NOTEPAD"
http://forums.parallax.com/forums/default.aspx?f=25&m=255771

Mirror releases current code for Gear. This is the current Gear thread including the stimulus plug-in. Also links backward to the original version.
http://forums.parallax.com/forums/default.aspx?f=25&m=242685

Oldbitcollector's PropBBS for Ethernet
http://forums.parallax.com/forums/default.aspx?f=25&m=257661

and now, FemtoBASIC in Color
http://forums.parallax.com/forums/default.aspx?f=25&m=231506

tvIRC client for propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=258073

Games

Spinpong
http://forums.parallax.com/forums/default.aspx?f=25&m=256210

Hardware

Beau troubleshoots his 3axis H48C module's prop code
http://forums.parallax.com/forums/default.aspx?f=25&m=225297

a different 3axis sensor: LIS3LV02DQ test program by JoMo (nice chip, no external adc needed, runs on 2v to 3.6v, qfn package though)
http://forums.parallax.com/forums/default.aspx?f=25&m=151211

position sensing, kicks around using a LVDT (Linear Variable Differential Transformer) to measure an physical movement.
http://forums.parallax.com/forums/default.aspx?f=25&m=255994
{ LVDT: [[http://en.wikipedia.org/wiki/Linear_variable_differential_transformer }]]
Also Beau points back to his July 31 2006 post for a homemade LVDT joystick, an overlooked wonder. Good stuff.
http://forums.parallax.com/forums/default.aspx?f=25&m=138059

towards implementing ADS8341 16 bit 4 channel A/D converter
http://forums.parallax.com/forums/default.aspx?f=25&m=257373

links to chinese caliper threads
http://forums.parallax.com/forums/default.aspx?f=25&m=255194

towards interfacing with vinculum vdrive 2 from FTDI to USB memory stick
http://forums.parallax.com/forums/default.aspx?f=25&m=257357

messing w/magnets (railguns and why one needs a blockhouse on range)
http://forums.parallax.com/forums/default.aspx?f=25&m=253606

on XBee modules, briefly. Link to Martin's site. Demo I/O programs added.
http://forums.parallax.com/forums/default.aspx?f=25&m=258257

Ping, briefly w/ links. Mike Green's ground plane discussion.
http://forums.parallax.com/forums/default.aspx?f=25&m=257936

managing ground traces
http://forums.parallax.com/forums/default.aspx?f=25&m=260436

Handy GPS search link for prop forum.
http://forums.parallax.com/forums/default.aspx?f=25&m=258918

tips for pulling protoboard prop chip
http://forums.parallax.com/forums/default.aspx?f=25&m=259005

shhh, let sleeping props lie: power down alternatives.
http://forums.parallax.com/forums/default.aspx?f=25&m=258976

about qfn chips. with Ti's qfn.pdf (nice!)
http://forums.parallax.com/forums/default.aspx?f=25&m=260015

technochip's Vinculum SPI, with spec sheet corrections.
http://forums.parallax.com/forums/default.aspx?f=25&m=260159

sinking 5v to the prop, discussion only no schematics.
http://forums.parallax.com/forums/default.aspx?f=25&m=259960

Ymodem: transfer files from prop's SD to a pc
http://forums.parallax.com/forums/default.aspx?f=25&m=260146

towards USB hosting. Nifty proposal: making a sniffer first.
http://forums.parallax.com/forums/default.aspx?f=25&m=260298

PCA9555 IO expander driver. W/link to deSilva's Small I2c Driver
http://forums.parallax.com/forums/default.aspx?f=25&m=260896

FTDI ComPort to Propeller: 2M baud, 3M still theoretical. (w/ link to FTDI app notes). No code yet, but one can hope. Exploits on Stephen's blog: http://propcandev.blogspot.com
Discursive prose about probs and props, yes! (de-obscurify: probs ~ problems)
http://forums.parallax.com/forums/default.aspx?f=25&m=260892

advice on making pcbs with cnc's, and alternatives. (w/ links.)
http://forums.parallax.com/forums/default.aspx?f=25&m=254578

Images & displays

text over camera image
http://forums.parallax.com/forums/default.aspx?f=25&m=259649

spudview: displays ppm images
http://forums.parallax.com/forums/default.aspx?f=25&m=259578

Tim's assembly graphics driver for µOLED-96-Prop (eeprom bin release). In Object Exchange.
http://forums.parallax.com/forums/default.aspx?f=25&m=258383
(Subtext: Beat this to win the contest.)

Resurgent threads

steppers in assembly. Alberto adds a driver.
http://forums.parallax.com/forums/default.aspx?f=25&m=241108

'Objects' module for PE Kit Labs -- hyperterminal alternative.
http://forums.parallax.com/forums/default.aspx?f=25&m=169419

new vid of Alberto's led curtain. (follow more-vids-from to see his mp3 player)
http://forums.parallax.com/forums/default.aspx?f=25&p=5&m=179824

Advanced

cogstop, coginit and cognew considerations. (Use cogstop....or not; nor CASE) and finally, add a wait so that a new stack settles down.
http://forums.parallax.com/forums/default.aspx?f=25&p=2&m=250812

assembly, towards programs bigger than cog memory. A rudimentary introduction to this topic but has links to LMM and prior threads
http://forums.parallax.com/forums/default.aspx?f=25&m=254117

notes on high speed sampling (25MHz), towards low rez vision. [no code, schematic, just theory].
sidebars: assembly 8x8 font routine (w/ bugs, more or less squashed), size & alignment issues
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=255764

Malloc memory manager, version 011.
http://forums.parallax.com/forums/default.aspx?f=25&m=256052

advanced skinny branches, Forth: Peter Jakacki offers up a droll implementation of Ladder Logic
http://en.wikipedia.org/wiki/Ladder_logic
used typically by Programmable Logic Controllers (PLC's)
http://en.wikipedia.org/wiki/Programmable_logic_controller
The discussion delves into the how-to but not so much the why-do-it.
http://forums.parallax.com/forums/default.aspx?f=25&m=256393

high frequency synthesis, with link back to a 2006 thread with Chip's teaser about a quickie microphone to fm transmitter (no specifics alas, shall we pester him?)
http://forums.parallax.com/forums/default.aspx?f=25&m=255074

towards a prop mp3 player (or just an UI frontend -- that's doable)
http://forums.parallax.com/forums/default.aspx?f=25&m=259258

TV object color limitations
http://forums.parallax.com/forums/default.aspx?f=25&m=258584

Prop VGA IRC Client
http://forums.parallax.com/forums/default.aspx?f=25&m=260624

adjusting video refresh rates with crystals
http://forums.parallax.com/forums/default.aspx?f=25&m=260446

towards FFT, DFT implementation. Links to past work.
http://forums.parallax.com/forums/default.aspx?f=25&m=259917

networking props. Links back to transfer speeds, Phil's reverse LMM.
http://forums.parallax.com/forums/default.aspx?f=25&m=260226

Fundamentalists

Skinning the onion -- any further and you're scraping hardware. Chip Gracey releases ROM source code. Booter, spin interpreter and runner.
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=252691

programming advanced, towards developing an independent spin compiler. Some solid ground here too, with fundamental text links. Sourceforge details too.
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=253050

spawning eeproms for the prop
http://forums.parallax.com/forums/default.aspx?f=25&m=257265

under development, Ale's Large Memory Model assembler
http://forums.parallax.com/forums/default.aspx?f=25&m=233324

Esoterica

CP/M operating system on a prop. Still handy. (links to reference sites)
http://forums.parallax.com/forums/default.aspx?f=25&p=2&m=252784

JVM (Java Virtual Machine) for the prop. Long thread documenting its implementation. Latest entries have best/stable/working code.
http://forums.parallax.com/forums/default.aspx?f=25&p=001&m=244721

England, props, importers and VAT, egads!
http://forums.parallax.com/forums/default.aspx?f=25&m=260775

Schedule

Prop Graphics Workshop at ATEA (April 2 - 4)
http://forums.parallax.com/forums/default.aspx?f=25&m=233234

uOLED-96-Prop design contest (April 30)
http://forums.parallax.com/forums/default.aspx?f=25&m=248327

2008 Prop design contest (September 1, 2008)
http://www.parallax.com/Default.aspx?tabid=603

IRC Saturdays at 17:00 gmt
http://forums.parallax.com/forums/default.aspx?f=25&m=252351

IRC Thursdays at 8pm to 10pm Eastern Australia (Canberra) time. (GMT+10?)
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=253060

Unofficial NE Prop Expo in Sandusky, Ohio (tentative Aug 22-23, 2008)
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=246525

Propeller Tool

Running multiple tools simultaneously now possible
http://forums.parallax.com/forums/default.aspx?f=25&m=255765

Propeller Tool 1.1 re-released. Near bulletproof prop programming.
http://forums.parallax.com/forums/default.aspx?f=25&m=244899

Prop Tool 1.1 problems
http://forums.parallax.com/forums/default.aspx?f=25&m=260734
http://forums.parallax.com/forums/default.aspx?f=25&m=260683
http://forums.parallax.com/forums/default.aspx?f=25&m=260801
linux flavored wine:
http://forums.parallax.com/forums/default.aspx?f=25&m=261123

DYI tales

automobile lcd (rca input) from ebay
http://forums.parallax.com/forums/default.aspx?f=25&m=249064

on Printed Circuit Boards, shops and tools (link to toner technique)
http://forums.parallax.com/forums/default.aspx?f=25&m=258070

Outside apps

character (cursor) bitmap encoder for Windows
http://forums.parallax.com/forums/default.aspx?f=25&m=257207

Luggable towtruck. Get one for your trunk today, because you never know
http://forums.parallax.com/forums/default.aspx?f=25&m=258479

STM32 Primer
http://forums.parallax.com/forums/default.aspx?f=25&m=259602

Projects

two player game console
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=251914

Alberto's homemade mp3 player. (using Vmusic2 chip) With code.
http://forums.parallax.com/forums/default.aspx?f=25&m=259540

Remote control props. With recursion sidebar discussion.
http://forums.parallax.com/forums/default.aspx?f=25&p=2&m=256003

Adverts

guitar effects pedal, Howler goes metal
http://forums.parallax.com/forums/default.aspx?f=25&m=252772

prop powered stickers
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=242622

PropGFX Lite DIP40 almost ready
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=248778

PropGFX: new modes
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=259544

almost here: ImageCraft C
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=254114

ImageCraft C alpha is here
http://forums.parallax.com/forums/default.aspx?f=25&m=260575

Prop Magazine - April 2008

last issue March 2008
next issue May 2008

Ramblings by the editor

Well, this is the 3rd edition of Prop Magazine. It won't be that long and it will be time for an anniversary. Speaking of anniversaries, this must be about the 2nd anniversary of the Propeller Chip (If someone knows the exact date can they drop a note in this thread please). So, I have decided to include some of the old posts in this edition. On a more recent note there have been a fair few object releases this month. Check out the new games, I2C slave, nice user interface on the Quad DRO and the others listed below.

The ESC was also this month and some of the people from Parallax were there. Check out their reports below. Also on show was the Coyote-1 and the danceBot.

History

A miracle chip?
http://forums.parallax.com/forums/default.aspx?f=25&m=110889

FullDuplexSerial is posted. Probably the most widely used object.
http://forums.parallax.com/forums/default.aspx?f=25&m=111614

Some early notes about the propeller.
http://forums.parallax.com/forums/default.aspx?f=25&m=113461

Propeller Guts - Some early documentation. Still helpful although most things are now in the manual and data sheet.
http://forums.parallax.com/forums/default.aspx?f=25&m=111215

Special Threads

Being April there were of course a couple of special threads. First up was the release of the Relleporp.
http://forums.parallax.com/forums/default.aspx?f=25&m=261265&g=261647#m261647

And secondly a hidden assembly instruction was found!
http://forums.parallax.com/forums/default.aspx?f=25&m=261282

Basics

Request for programs to put in a kiosk program for ESC (with any luck they will release what they used, get the hint?)
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=262904
http://forums.parallax.com/forums/default.aspx?f=33&m=262907

Questions about how long it takes to start a cog and other ways of accomplishing things.
http://forums.parallax.com/forums/default.aspx?f=25&m=263541&g=263593

Sine wave output from the prop
http://forums.parallax.com/forums/default.aspx?f=25&m=261056&g=261067

Pre-fetch and self modifying code. Something to watch out for!
http://forums.parallax.com/forums/default.aspx?f=25&m=260965

Some questions and answers on accessing bytes in larger variables.
http://forums.parallax.com/forums/default.aspx?f=25&m=262787

Some general questions and answers about what the prop can do.
http://forums.parallax.com/forums/default.aspx?f=25&m=264271

Getting data from one pin to another and a how to use the timers to do it.
http://forums.parallax.com/forums/default.aspx?f=25&m=264649

Changing variables between long and byte.
http://forums.parallax.com/forums/default.aspx?f=25&m=265586

Basics and Beyond

A question about required crystal accuracy.
http://forums.parallax.com/forums/default.aspx?f=25&m=261089&g=261115

and more questions about crystals
http://forums.parallax.com/forums/default.aspx?f=25&m=259385&g=264662
http://forums.parallax.com/forums/default.aspx?f=25&m=264577&g=264843

What are the threshold voltages of the prop? This tells you for one prop.
http://forums.parallax.com/forums/default.aspx?f=25&m=260930&g=261152

Questions and answers about what can hang a cog
http://forums.parallax.com/forums/default.aspx?f=25&m=260908&g=261255

Another fried PLL and some hints from Paul about how prevent it happening.
http://forums.parallax.com/forums/default.aspx?f=25&m=261542&g=262166

Pops and clicks in audio and how to avoid them.
http://forums.parallax.com/forums/default.aspx?f=25&m=261767

Tools and methods for fixing memory corruption problems.
http://forums.parallax.com/forums/default.aspx?f=25&m=253901

Using the pos/neg edge detector timer modes.
http://forums.parallax.com/forums/default.aspx?f=25&m=262573

Initial states for the c and z flags.
http://forums.parallax.com/forums/default.aspx?f=25&m=262837&g=263003

QTI sensor code is here.
http://forums.parallax.com/forums/default.aspx?f=25&m=263689&g=263689

What does the @ operator do in asm? Also a tip on the @@ operator.
http://forums.parallax.com/forums/default.aspx?f=25&m=264399

Questions about graphics and VGA (again, hopefully some answers this time).
http://forums.parallax.com/forums/default.aspx?f=25&m=264361&p=1

More discussion about high speed comms.
http://forums.parallax.com/forums/default.aspx?f=25&m=264379&g=264477

Problems with the CORDIC object.
http://forums.parallax.com/forums/default.aspx?f=25&m=264018

Questions about SPIN speed and the interpreter.
http://forums.parallax.com/forums/default.aspx?f=25&m=266040&g=266043#m266043

Advanced

Creative use of the object[x].method notation to solve some problems
http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=260504

Questions about queues and some examples
http://forums.parallax.com/forums/default.aspx?f=25&m=262504

Discussions on loading new LMM cogs in ImageCraft C
http://forums.parallax.com/forums/default.aspx?f=25&m=263048&g=263456

Discussions about new features for the spin compiler.
http://forums.parallax.com/forums/default.aspx?f=25&m=260170&g=261362

Using the prop as an RF transmitter and receiver.
http://forums.parallax.com/forums/default.aspx?f=25&m=261275

New Programs and Objects

An I2C Slave driver is now available in spin and assembly thanks to hippy (who has a D'Oh moment while testing multiple slaves).
http://forums.parallax.com/forums/default.aspx?f=25&m=263375&g=263520

Distance and Bearings calculations object for the prop by Chuck.
http://forums.parallax.com/forums/default.aspx?f=25&m=263181&g=263266#m263266

Ymodem upload utility by Rayman. Upload files from your propeller to your PC!
http://forums.parallax.com/forums/default.aspx?f=25&m=261220

High speed serial by Stephen.
http://forums.parallax.com/forums/default.aspx?f=25&m=262414&g=263008

Quad DRO by Richard. Useful if you have a mill/lathe or anything else you need measurement info for.
http://forums.parallax.com/forums/default.aspx?f=25&m=262412&g=263235

A pop3 email client for the prop that displays the message From: and Subject: fields on your TV.
http://obex.parallax.com/objects/259/

On prop development using a spin like syntax. Again thanks to hippy.
http://forums.parallax.com/forums/default.aspx?f=25&m=264289&g=264403

Steven finally finishes the XOR version of Graphics.spin including the ROM font.
http://forums.parallax.com/forums/default.aspx?f=25&m=248864&g=264881#m264881

PropBASH, a modded version of PropDOS that is more like BASH
http://forums.parallax.com/forums/default.aspx?f=25&m=265024

VGA Text driver for ImageCraft C.
http://forums.parallax.com/forums/default.aspx?f=25&m=266216

Gotcha

Assembly, MUL details divulged. (w/commented code)
http://forums.parallax.com/forums/default.aspx?f=25&m=261282

References

Basics

Prefetch impacts self-modifying code (w/link to deSilva asm tutorial)
http://forums.parallax.com/forums/default.aspx?f=25&m=260965

Can a cog get stuck waiting?
http://forums.parallax.com/forums/default.aspx?f=25&m=260908

byte basics
http://forums.parallax.com/forums/default.aspx?f=25&m=262787

Basics and Beyond

Audio: pops at start and stop of wav file
http://forums.parallax.com/forums/default.aspx?f=25&m=261767

Debugging programs, best practices (divide and conquer)
http://forums.parallax.com/forums/default.aspx?f=25&m=253901

On negative edge counting (counter setups)
http://forums.parallax.com/forums/default.aspx?f=25&m=262573

Advanced

On queues
http://forums.parallax.com/forums/default.aspx?f=25&m=262504

GPS: conversions, ddmm.mmmm to dd.dddd, radians. (link to Circuit Cellar)
http://forums.parallax.com/forums/default.aspx?f=25&m=262819

Parallax stuff

embedded props get a write up.

http://forums.parallax.com/forums/default.aspx?f=25&m=261005

propplug logo up on protoboard
http://forums.parallax.com/forums/default.aspx?f=25&m=262269

Hardware

on radio control,
http://forums.parallax.com/forums/default.aspx?f=25&m=262102

signals between 3.6v and 5v chips
http://forums.parallax.com/forums/default.aspx?f=25&m=262865

Counters

FSK, signal synthesis, decoding discussion.
http://forums.parallax.com/forums/default.aspx?f=25&m=261275

Demos

Ymodem: upload data as [binary] file
http://forums.parallax.com/forums/default.aspx?f=25&m=261220

Circuitry

circuit design precautions and guidelines (how2 stop frying pll's)
http://forums.parallax.com/forums/default.aspx?f=25&m=261542&p=1

use a 1K pull-up on prop plug whose cable will removed
http://forums.parallax.com/forums/default.aspx?f=25&m=261253
more info inc fish's schematic:
http://forums.parallax.com/forums/default.aspx?f=25&m=192068

on pull-up resistors
http://forums.parallax.com/forums/default.aspx?f=25&m=261993

Adverts

Imagecraft C 20% discount
http://forums.parallax.com/forums/default.aspx?f=25&m=261910

Prop Magazine - May 2008

last issue April 2008
next issue June 2008

Ramblings by the editor

Welcome to the 4th edition of Prop Magazine. Everyone is welcome to create new sections and add new links.; there is no need to login to or register do so.

Events

Oldbitcollector's Unofficial Propeller expo NE: SATURDAY - AUGUST 23RD
http://forums.parallax.com/forums/default.aspx?f=25&m=246525&p=3

Software

Propellent Library and Executable - standalone Windows SPIN and ASM compiler
http://forums.parallax.com/forums/default.aspx?f=25&m=270747

Getting two copies of the PropTool IDE running simultaneously
http://forums.parallax.com/forums/default.aspx?f=25&m=267915

Reducing the size of the VGA drivers
http://forums.parallax.com/forums/default.aspx?f=25&m=267605

Latest ImageCraft ICCV7 information
http://forums.parallax.com/forums/default.aspx?f=25&m=267452

TV Text Driver for ImageCraft C compiler
http://forums.parallax.com/forums/default.aspx?f=25&m=270075

FullDuplexSerial Driver for ImageCraft C compiler
http://forums.parallax.com/forums/default.aspx?f=25&m=269905

VGA Text Driver for ImageCraft C compiler
http://forums.parallax.com/forums/default.aspx?f=25&m=266216

Mouse Driver for Imagecraft C compiler
http://forums.parallax.com/forums/default.aspx?f=25&m=267752

ImageCraft : CLIB COG and how to do a CMPEXCH?
http://forums.parallax.com/forums/default.aspx?f=25&m=270933

Virtual Interrupts using LMM
http://forums.parallax.com/forums/default.aspx?f=25&m=266441

How to read strings representing floating point numbers
http://forums.parallax.com/forums/default.aspx?f=25&m=266569

Some info on BB_FullDuplexSerial
http://forums.parallax.com/forums/default.aspx?f=25&m=265580

Library code for using a DS1307 and 32-bit Unix Time
http://forums.parallax.com/forums/default.aspx?f=25&m=232817

Numeric Conversion Program!
http://forums.parallax.com/forums/default.aspx?f=25&m=271093

Integer Square Roots
http://forums.parallax.com/forums/default.aspx?f=25&m=269640

Handling DS1631 I2C temperature sensor data
http://forums.parallax.com/forums/default.aspx?f=25&m=270815

GameCube Driver
http://forums.parallax.com/forums/default.aspx?f=25&m=270337

SD Card File Copy
http://forums.parallax.com/forums/default.aspx?f=25&m=268224

Bugs in MIDI Object
http://forums.parallax.com/forums/default.aspx?f=25&m=269919

Hardware

USB Powered ProtoBoard
http://forums.parallax.com/forums/default.aspx?f=25&m=268412

Using LDO with 3.7V LiPo battery
http://forums.parallax.com/forums/default.aspx?f=25&m=267584

Creating a lightpen for the Propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=267885

Getting TV input working
http://forums.parallax.com/forums/default.aspx?f=25&m=267011

Getting Servo32 up and running
http://forums.parallax.com/forums/default.aspx?f=25&m=267973

Latest report on AiChip Industries I2C Slave
http://forums.parallax.com/forums/default.aspx?f=25&m=263375

Updated discussion on using IDE drives with a propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=246610

Interfacing I2C 3V3 to 5V
http://forums.parallax.com/forums/default.aspx?f=25&m=262865&p=3

Interfacing SD Card using the VGA connector
http://forums.parallax.com/forums/default.aspx?f=25&m=267538

Some questions on hooking up an SD Card
http://forums.parallax.com/forums/default.aspx?f=25&m=266862

Getting PINK and XBee working with the Propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=267342

Motor control with the Propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=267202

Trials and tribulation in getting RS232 comms working
http://forums.parallax.com/forums/default.aspx?f=25&m=266245

Using the Propeller as a PS/2 device
http://forums.parallax.com/forums/default.aspx?f=25&m=267553

DTMF generation and detection
http://forums.parallax.com/forums/default.aspx?f=25&m=266381

Damaged I/O pins
http://forums.parallax.com/forums/default.aspx?f=25&m=268573
http://forums.parallax.com/forums/default.aspx?f=25&m=268602

Multiple PWM Outputs
http://forums.parallax.com/forums/default.aspx?f=25&m=270722

Powering Propeller from higher voltages
http://forums.parallax.com/forums/default.aspx?f=25&m=270832

Battery Monitoring
http://forums.parallax.com/forums/default.aspx?f=25&m=270848

ADC0838 as DVM for Propeller
[http://forums.parallax.com/forums/default.aspx?f=25&m=270062]]

Prop Tool

Prop Tool on Asus EEE PC
http://forums.parallax.com/forums/default.aspx?f=25&m=270368

Prop Tool and documentation version numbering mis-matches
http://forums.parallax.com/forums/default.aspx?f=25&m=260734
http://forums.parallax.com/forums/default.aspx?f=25&m=270442

Prop Tool Line Continuation / Multi-Line Statements
http://forums.parallax.com/forums/default.aspx?f=25&m=269758

Prop Tool file locating
http://forums.parallax.com/forums/default.aspx?f=25&m=269896

Propeller Download via XBee
http://forums.parallax.com/forums/default.aspx?f=25&m=270015

Propeller Download without Reset connection
http://forums.parallax.com/forums/default.aspx?f=25&m=270355

Resources and Learning

Discussion and resources on learning about digital logic
http://forums.parallax.com/forums/default.aspx?f=25&m=266117

Discussion about the lack of interrupts on the Propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=266423

An explanation why the Prop doesn't have on-chip Eeprom
http://forums.parallax.com/forums/default.aspx?f=25&m=266601

WAITPEQ and WAITPNE timing
http://forums.parallax.com/forums/default.aspx?f=25&m=267689

Starting Cogs with an arbitrary number of parameters
http://forums.parallax.com/forums/default.aspx?f=25&m=267499

Launching multiple Cogs
http://forums.parallax.com/forums/default.aspx?f=25&m=268862

Inter-Cog communications
http://forums.parallax.com/forums/default.aspx?f=25&m=266026

Fast timing / short waits in Spin
http://forums.parallax.com/forums/default.aspx?f=25&m=268748

High-Temperature Sensors
http://forums.parallax.com/forums/default.aspx?f=25&m=268702

Discussion on UTF-8 versus UTF-32 in the Propeller Tool and in general
http://forums.parallax.com/forums/default.aspx?f=25&m=268608

Parallel Processing
http://forums.parallax.com/forums/default.aspx?f=25&m=269360

I2C Slave Addresses
http://forums.parallax.com/forums/default.aspx?f=25&m=270079
http://forums.parallax.com/forums/default.aspx?f=25&m=269595

Mixing Spin and Assembler
http://forums.parallax.com/forums/default.aspx?f=25&m=270347
http://forums.parallax.com/forums/default.aspx?f=25&m=270698

Random thought for Prop II: Separate execution units for hub ops.
http://forums.parallax.com/forums/default.aspx?f=25&m=270924

Spin Start/Stop Methods
http://forums.parallax.com/forums/default.aspx?f=25&m=270964

USB with the Propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=269677

Propeller current consumption
http://forums.parallax.com/forums/default.aspx?f=25&m=270041

Physical contact sensing alternatives
http://forums.parallax.com/forums/default.aspx?f=25&m=271136

Fun and Games

Rayman's Ladder Game
http://forums.parallax.com/forums/default.aspx?f=25&m=267604

The Propeller-based Space Invaders Clock
http://forums.parallax.com/forums/default.aspx?f=25&m=266560

The Propeller as an Art Form
http://forums.parallax.com/forums/default.aspx?f=25&m=266855
http://forums.parallax.com/forums/default.aspx?f=25&m=266710

Predicting the future with a Propeller Chip
http://forums.parallax.com/forums/default.aspx?f=25&m=268823

The Pixelmusic 3000 on Make 14
http://forums.parallax.com/forums/default.aspx?f=25&m=270787

Projects

Ideas for project sought
http://forums.parallax.com/forums/default.aspx?f=25&m=266703

Persistence of Vision
http://forums.parallax.com/forums/default.aspx?f=25&m=267954

Quad DRO, and a request for beta testers
http://forums.parallax.com/forums/default.aspx?f=25&m=262412

The Incredible Message Machine
http://forums.parallax.com/forums/default.aspx?f=25&m=267679

A nice and compact LED Matrix display
http://forums.parallax.com/forums/default.aspx?f=25&m=267021

Latest news on the OpenStomp(TM) Coyote-1 (Guitar Effects Pedal)
http://forums.parallax.com/forums/default.aspx?f=25&m=252772&p=3

NAVCOM AI UAV/UMV code and schematics
http://forums.parallax.com/forums/default.aspx?f=25&m=190494

An on-screen function plotter
http://forums.parallax.com/forums/default.aspx?f=25&m=268700

Some thoughts on emulating a PICmicro 18F84
http://forums.parallax.com/forums/default.aspx?f=25&m=266879

Vision for the Propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=268814

Juicebox Weather Station
http://forums.parallax.com/forums/default.aspx?f=25&m=269246

Propeller-based datalogger
http://forums.parallax.com/forums/default.aspx?f=25&m=268880

New OS - PorthOS is born
http://forums.parallax.com/forums/default.aspx?f=25&m=269867

Competitions

Results of uOLED-96-PROP Object Design Contest
http://forums.parallax.com/forums/default.aspx?f=25&m=248327

Propeller Products

New ProtoBoard with USB on-board on its way
http://forums.parallax.com/forums/default.aspx?f=25&m=269025

Parallax Propellent Downloader Library and Executable v1.0
http://forums.parallax.com/forums/default.aspx?f=25&m=270747
http://forums.parallax.com/forums/default.aspx?f=25&m=268343

The new Parallax Serial Terminal (PST)
http://forums.parallax.com/forums/default.aspx?f=25&m=267427

New Propeller Videos and Audio Podcast
http://forums.parallax.com/forums/default.aspx?f=25&m=268524

Upgrading the Eeprom on a PropStick
http://forums.parallax.com/forums/default.aspx?f=25&m=265409

What to buy
http://forums.parallax.com/forums/default.aspx?f=25&m=267645

Choosing between Spin Stamp and Propeller

http://forums.parallax.com/forums/default.aspx?f=25&m=267071

Buying and Shipping
http://forums.parallax.com/forums/default.aspx?f=25&m=267606

Wanting a Parallax Quad Rover
http://forums.parallax.com/forums/default.aspx?f=25&m=269525

Propeller Mk I with 64 Pin I/O update
http://forums.parallax.com/forums/default.aspx?f=25&m=270526

Adverts

New third-party Propeller board
http://forums.parallax.com/forums/default.aspx?f=25&m=268789

USB FTDI bare PCB's for sale
http://forums.parallax.com/forums/default.aspx?f=25&m=269145

PropSTICK Kit Bare Printed Circuit Boards
http://forums.parallax.com/forums/default.aspx?f=25&m=271072

New Sparkfun Protoboard
http://forums.parallax.com/forums/default.aspx?f=25&m=270979

A case (box) for the Propeller
http://forums.parallax.com/forums/default.aspx?f=25&m=269900

Prop Magazine - June 2008

last issue: May 2008

Ramblings by the editor

Welcome to the 5th edition of Prop Magazine. Everyone is welcome to create new sections and add new links.; there is no need to login to or register do so.

Events

Oldbitcollector's Unofficial Propeller expo NE: SATURDAY - AUGUST 23RD
http://forums.parallax.com/forums/default.aspx?f=25&m=246525&p=3

Software

Fryview (based on Spudview idea) image viewer and converter released,
http://forums.parallax.com/forums/default.aspx?f=25&m=272283

Hardware

Prop Tool

Propeller Tool 1.2 Released
http://www.parallax.com/tabid/442/Default.aspx

Parallax Propellent Library and Executable v1.0
http://forums.parallax.com/forums/default.aspx?f=25&m=270747

Resources and Learning

Discussion about connecting PSX controllers
http://forums.parallax.com/showthread.php?104124-Playstation-Controller-Guitar-Hero

Fun and Games

New Game: Jumping Jack
http://forums.parallax.com/showthread.php?104089-New-Game-Jumping-Jack

Projects

Google Earth GPS SD Card Logger
http://forums.parallax.com/showthread.php?104049-Google-Earth-GPS-SD-Card-Logger-COMPLETE!

Competitions

Propeller Products

Adverts

Current Beta API:

```
  Ethernet TCP/IP Socket Layer Driver (IPv4)
  ------------------------------------------

  Copyright (C) 2006 - 2007 Harrison Pham

  This file is part of PropTCP.

  PropTCP is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 3 of the License, or
  (at your option) any later version.

  PropTCP is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>
.

Object "driver_socket" Interface:

PUB  start(cs, sck, si, so, int, xtalout, macptr, ipconfigptr) : okay
PUB  stop
PUB  listen(port)
PUB  connect(ip1, ip2, ip3, ip4, remoteport, localport)
PUB  close(handle)
PUB  isConnected(handle)
PUB  isValidHandle(handle)
PUB  readByteNonBlocking(handle) : rxbyte
PUB  readByte(handle) : rxbyte
PUB  writeByteNonBlocking(handle, txbyte)
PUB  writeByte(handle, txbyte)
PUB  resetBuffers(handle)

Program:   1,798 Longs
Variable:    131 Longs
```

```
PUB  start(cs, sck, si, so, int, xtalout, macptr, ipconfigptr) : okay
```

```
 Call this to launch the Telnet driver
 Only call this once, otherwise you will get conflicts
 macptr      = HUB memory pointer (address) to 6 contiguous mac addres
s bytes
 ipconfigptr = HUB memory pointer (address) to ip configuration block
(20 bytes)
               Must be in order: ip_addr. ip_subnet, ip_gateway, ip_dn
s


_________
PUB  stop

 Stop the driver


____________________
PUB  listen(port)

 Sets up a socket for listening on a port
 Returns handle if available, -1 if none available
 Nonblocking


_____________________________________________________________
PUB  connect(ip1, ip2, ip3, ip4, remoteport, localport)

 Connect to remote host
 Returns handle to new socket, -1 if no socket available
 Nonblocking


____________________
PUB  close(handle)

 Closes a connection


_____________________________
PUB  isConnected(handle)

 Returns true if the socket is connected, false otherwise


_______________________________
PUB  isValidHandle(handle)

 Checks to see if the handle is valid, handles will become invalid onc
e they are used
 In other words, a closed listening socket is now invalid, etc


_____________________________________________
```

```
PUB  readByteNonBlocking(handle) : rxbyte

 Read a byte from the specified socket
 Will not block (returns -1 if no byte avail)


_____________________________________
PUB  readByte(handle) : rxbyte

 Read a byte from the specified socket
 Will block until a byte is received


_____________________________________________
PUB  writeByteNonBlocking(handle, txbyte)

 Writes a byte to the specified socket
 Will not block (returns -1 if no buffer space available)


_____________________________________
PUB  writeByte(handle, txbyte)

 Write a byte to the specified socket
 Will block until space is available for byte to be sent


_______________________________
PUB  resetBuffers(handle)

 Resets send/receive buffers for the specified socket
```

Pulse Width Modulation

The topic of PWM is complex - and trivial at the same time.
When you have the need to dim an LED connected to I/O = *ledPin* between *dimPercent* = 0 to 99%, do just this:

```
DIRA[ledPin] := 1
CTRA := %0_0110<<26 + ledPin
FRQA := $7FFF_FFFF/50 * dimPercent
```

In the following sections we shall discuss

1. What is PWM in the first place?
2. How to control a PWM channel with SPIN
3. How to control two PDM channels using the timers/counters
4. How to control A LOT of PWM channels
5. Adding a low-pass filter makes a DAC!

... but maybe AFTER Xmas..

What is PWM in the first place?

Someone "doing PWM" will generate some of the signals shown in the following sketch. You notice that they are digital wrt ("with respect to") amplitude. They can also be discrete wrt the time axis. A signal transports information. The information with PWM ("pulse *width modulation*") is coded into the relative length of the pulse wrt the period.
The quotient between both is called *duty cycle*, which is a value between 0 and 100% (or between 0 and 1, when you are using REAL numbers)

This is the definition "by the book". Note the beauty of it: It is not only independent of the amplitude, making it immune against typical "noise" effects, but also independent of the choice of the period, so avoiding a strict synchronization of clock speeds.

In most situations however the period is a fixed design parameter, giving the pulse width (measured in absolute units) a meaning of its own. So you speak of a "2ms pulse" when driving servos, independent of the "duty cycle" or period.

In both cases the receiver can have problems with duty cycles of exactly 0 or 100%, so theses values should be avoided.

So the information transported consists of a series of values (the "duty cycles", or the absolute "pulse

widths") within (generally) fixed length time slots (= each "period"). Such kind of information can also be transported in many other ways:

- as (P)AM ("amplitude modulation"): The value of the signal voltage is modified - this is what we are accustomed to.
- as (P)FM ("frequency modulation"): The number of on/off signals within the time slot (the "frequency") is modified - old-fashioned *modems* worked that way.
- as PPM ("phase modulation"): The exact location of a small spike within the time slot is modified - this is very related to PWM, a differentiated PWM signal as it were.
- as PDM ("density modulation"): A certain number of **fixed length** pulses ("spikes" again!) is arranged within the time slot (=period). This is quite similar to PWM (where all those pulses are assembled at one side of the period, so to speak) but not exactly the same. It is the way we used the DUTY-mode of the Propeller timer/counter in the first example above.
  A PDM signal is generally interpreted as a Bitstream of "ones" and "zeroes"; the decoded value being the (moving) average.

Each of those modulations has its pros and its cons wrt to noise immunity, ease of transmission ("encoding"), ease of receiving ("decoding"), cost,....

In "microcontrolling" we use PWM (and also PDM) mainly for three reasons.

1. To control a **servo** (period: 20ms, pulse width: 1 .. 2ms)
2. To control the average current transported to an **inert device** (light bulb, motor, the system LED/human eye,...)
3. **DAC**: To generate a duty cycle proportional **voltage**, utilizing an appropriate low-pass filter (period

How to control a PWM channel with SPIN

So "hands-on"! An LED is such a useful device; one should invent it if it wasn't already there. The brightness of an LED is roughly proportional to the current flowing through it. According to this current the LED controls its voltage drop. It is always close to the forward voltage, a little bit higher with high current, a little bit lower with low current.

We can easily construct a Current-DAC by - say - connecting the LED with three Propeller pins, each with a different resistor, as 220 Ohms, 470 Ohms, 1k, e.g. So we can readily provide different currents (20 mA ... 2mA), but at the cost of three pins.

The more common solution however is to **pulse** the LED, e.g. 50 ms off, 50 ms on, using the full output of one single I/O pin. The signal will look exactly as the middle situation in the above sketch. The LED will now shine with half its intensity. So we hope.

```
DIRA[0] := 1
dim_try1(0,50)                          ' LED is connected to I/O 0 '


PUB dim_try1(ledPin, dutyCycle) | onePercentTick, deadline
' version 3.1 2007 by deSilva '
' Dims an LED, dutyCycle is 0 to 100, ledPin is 0 to 31 '

  IF dutyCycle =< 0
     OUTA[ledPin] := 0
     RETURN
  IF dutyCycle =>100
     OUTA[ledPin] := 1
     RETURN

  deadline := CNT
  onePercentTicks := CLKFREQ/1000      ' 1ms = 1%'
  REPEAT
    deadline += onePercentTicks*100   ' 100 ms loop'
    WAITCNT(deadline)
    OUTA[ledPin] := 1
    WAITCOUNT(deadline + onePercentTicks*dutyCycle)
    OUTA[ledPin] := 0
```

Hey, it works! However...
(a) It flickers!
(b) The routine DIM_TRY1 never returns!

The period has to be adjusted to the application! We are sending zeros and ones - so we "see" zeros and ones. To trick our eyes we have to be faster, like a true magician. Have you already spotted the relevant parameter? Right, it's 1000, setting 1% of a period to 1ms (thus the period to 100 ms). We can readily change this to 10_000, making the period 10 ms which will suffice to do the trick.

Can we make it even faster? Try it out!
There are limits with SPIN: We are waiting for a time gap of "onePercentTicks" for a 1% dutyCyle; this must be >800, which means around a 1 kHz period. Sorry folks, that's simple SPIN . We will do much better soon, with a little help from a timer/counter.

It is typical for Propeller programming to have routines that can never return, as they have to be on the alert for the environment. We often call those routines "drivers". Other microcontrollers use "Interrupts" for this; the Propeller way is to engage a COG.

```
SUB main | dutyCycle, someStack[20]
  COGNEW(dim_try2(0, @dutyCycle), @someStack)
```

```
  REPEAT
    REPEAT WHILE ++dutyCycle <100
      WAITCNT(CNT+CLKFREQ/100)
    REPEAT WHILE --dutyCycle >0
      WAITCNT(CNT+CLKFREQ/100)
```

We have to make some modifications to our routine:

- In a fresh new COG the I/O characteristics have to be set again (DIRA in this case)
- We now provide an address rather than a value for the dutyCycle parameter
- We have to care for the occurrence of "bad values" within the main loop

```
PUB dim_try2(ledPin, dutyCycleAddr) | onePercentTicks, deadline,
 dutyCycle
' version 4.0 2007 by deSilva '
' Dims an LED, dutyCycle is 0 to 100, ledPin is 0 to 31 '

  DIRA[ledPin] := 1
  deadline := CNT
  onePercentTicks := CLKFREQ/10_000      ' 100µs = 1% '
  REPEAT
    dutyCycle := 0 #> LONG[dutyCycleAddr] <# 100
    deadline += onePercentTicks*100   ' 10 ms loop'
    WAITCNT(deadline)
    IF dutyCycle
       OUTA[ledPin] := 1
       IF dutyCycle < 100
          WAITCOUNT(deadline + onePercentTicks*dutyCycle)
          OUTA[ledPin] := 0
```

---> I left a BUG here, for the gentle reader to spot :-)

Now, onto the last part of this section. Isn't it a shame that we have to constrain ourselves to pulses of at most 10 µs within a period of 1 ms? The main issue will most likely be not the period, but the accuracy of the pulse length. Having 100 choices only would mean an angular accuracy of 3.6° for a servo.
Do we really need assembly language to overcome that?

Not at all! There is something much better in every COG: a **timer** (even two A and B - see next section). The working of the timers is thoroughly explained in Parallax AN001 (and most likely somewhere here in the wiki soon...link?)
It's simplicity itself: In their **NCO mode**

- A timer keeps adding the FREQ register to the PHS register each and every tick.

- The MSB (bit 31) of the PHS becomes connected to one of the I/O pins.

Thats all, really! Think some time what different things you can do with this!

The PWM algorithm then goes like this:

- Set FRQ to 1
- Set PHS to the negative number of ticks for your pulse
- Start the timer
- Do this in a loop of the length of your period

.. and here comes the code..

```
PUB dim_try3(ledPin, dutyCycleAddr) | onePercentTicks, deadline,
 dutyCycle
' version  2007 by deSilva '
' Dims an LED using TMRA, dutyCycle is 0 to 100, ledPin is 0 to 31 '

  DIRA[ledPin] := 1
  deadline := CNT
  onePercentTicks := CLKFREQ/10_000      ' 100µs = 1% '
  REPEAT
       dutyCycle := 0 #> LONG[dutyCycleAddr] <# 100
       WAITCNT(deadline += onePercentTicks*100)   ' 10 ms loop'
' ---- this part contains the improvement using a timer:
          '
'  Programming timerA for PWM-mode ("NCO")
          '
'  i.e. pulse = sign bit (bit 31); thus preset register with MINUS pul
se width  '
       CTRA := 0      ' reset timer                    '
       FRQA := 1      ' adding 1 @ system clock = 80 MHz'
       PHSA := -(onePercentTicks * dutyCycle)
       CTRA :=  (%0_00100 << 26) + ledPin
' Now there is a all the time of the world to do other things ....
          '
' Note that it is not very critical as the pulse is reset automaticall
y!        '
```

How to control two PDM channels using the timers/counters

(...coming soon)
In the meantime have a look at Martin Hebels **BS2 functions**. This is exactly what we did in the intro

example

How to control A LOT of PWM channels

(...coming soon)
In the meantime please refer to some forum-threads, e.g.
http://forums.parallax.com/forums/default.aspx?f=25&m=227109&g=229281

Adding a low-pass filter makes a DAC!

(to be continued...)

RCTIME Object

by Beau Schwabe

This object measures the time constant of an RC circuit. You can use RCTIME to read a variable resistance such as a potentiometer (e.g., volume knob or joystick), photosensor (CdS cell), thermistor (temperature), etc.

This object can operate in one of two modes:
• Background mode, in which a separate process (cog) asynchronously updates a variable.
• Foreground mode, in which a method call performs a "one-shot" time constant measurement.

RCTIME in foreground mode functions much like the Basic Stamp's RCTIME command.

Hardware

Build your variable resistance into a circuit such as this:

RC circuit (Type I)
RCTIME charges the capacitor to Vdd (the 200 ohm resistor limits the charging current to protect the Propeller's I/O pin) and then measures the time it takes for the capacitor to discharge through the variable resistance to Vdd/2.

You can also wire up the circuit as follows:

RC circuit (Type II)
For this second type of circuit, RCTIME first discharges the capacitor and then measures the time for it to charge to Vdd/2.

The first circuit is recommended as it provides slightly higher resolution due to the I/O threshold not being exactly Vdd/2.

Theory of operation

The results that RCTIME provides are proportional to the variable resistance R, not a direct measurement of R. The time to discharge (or charge) to Vdd/2 is given by this equation:

$$t = R\ C\ \ln 2$$

where t is the charging time in seconds, R is the resistance in ohms, and C is the capacitance in farads.

The result returned by RCTIME is t x clkfreq / 16 (RCTIME divides by 16 to eliminate the noisy least significant bits).

Example

Assume a resistance that can vary between 1000Ω and 2000Ω and a fixed capacitance of 0.1µF. At the low end (R = 1000Ω), the discharge time is 69.3µs and the result given by RCTIME is 347 (assuming an 80MHz system clock).

At the high end (R = 2000Ω), the discharge time is 139µs and the result given by RCTIME is 693.

Therefore, you would expect readings to vary over the range of the resistance from 347 to 693 (approximately; real numbers will of course differ from the calculated values).

Using RCTIME in background mode

To use RCTIME in background mode, call the **`start`** method with the following arguments:

1. pin: the number of the Propeller's I/O pin that is attached to the RC circuit
2. state: this argument should be 1 for Type I circuits, 0 for Type II.
3. variable address: the address of a long variable that is to be updated.

Example:

```
' ... clock definitions go here...
obj RC: "RCTIME"
var long pot_position
pub main
RC.start( 5, 1, @pot_position )
repeat
    ' here do something with pot_position, knowing that RCTIME is upda
ting it
    ' automatically in the background with new values
```

Note that the variable (in this example, **`pot_position`**) will only be updated as fast as the circuit discharges (or charges). You can monitor more than one pin, but keep in mind that background mode consumes an additional cog for each pin being monitored.

Using RCTIME in foreground mode

To use RCTIME in foreground mode, call the **`RCTIME`** method to initiate a measurement. Pass the same arguments that you would pass to **`start`**:
1. pin: the number of the Propeller's I/O pin that is attached to the RC circuit.
2. state: this argument should be 1 for Type I circuits, 0 for Type II.
3. variable address: the address of a long variable that will contain the result.

Example:

```
' ... clock definitions go here...
obj RC: "RCTIME"
var long pot_position
pub main
repeat
    RC.RCTIME( 5, 1, @pot_position ) ' Read pot position
    ' here do something with pot_position
```

Note: You cannot mix background and foreground operation in the same program.

Download

http://obex.parallax.com/objects/276/

Some calculations of the spin code SIZE to reference 'global' variables in various ways. Speed was not tested.

For reference, when variables are the same object. Referencing a stack variable and object VAR 1 byte. Referencing a DAT variable 2 bytes. Referencing a memory location from a Constant pointer was 4 bytes.

For variables shared from another object there is normally a 2 step process, retrieve the pointer and then retrieve the variable using the pointer. Because of the large difference in how a variable is accessed seen above, the storage of the pointer becomes key. If used in only one location, the best method is to retrieve the address on initialisation, and store with either in a object word VAR. If used more than once in the same method, store the pointer in a stack variable. These 2 techniques use 3 and 2 bytes to dereference respectively, and the code setting up the pointer is similarly sized. Once the extra 2 byte to store a stack pointer (long) vs a VAR pointer (word) is taken into account they are the same size. However the local var method is prefered as is expected to be faster as less code is loaded into the cog in the loop.

If you are using a pointer to iterate an array, there is significant difference depending how the array offsets are handled. The most effective is to use the offset array syntax, as the increment code is far smaller, except for BYTE arrays. So use blah := WORD/LONG/BYTE[stackPtr][stackOffset++], where both the ptr and offset are stored in stack variables.

Several software projects available for the Propeller.

- A list of playable games.
- A list of graphics drivers.
- Object Reference - in the same format as the Propeller Object Exchange

RFID with simple hardware

Micah Dowty has produced what might be "the world's simplest RFID reader design" using a Propeller Chip and just a few passive components, no pre-built RFID receiver / interface module required.

Details of the hardware plus an explanation of operation and associated software can be found in the Propeller Forum : here

The propeller is unusual in that it can generate a video signal in software. There is hardware support, but you need to have software constantly feeding that hardware with pixel and sync data. A complete video frame's worth of data 60 times a second for NTSC (50 for PAL).

The simplest graphics drivers consist of a single cog which both prepares the data for display and sends it to the video hardware. Examples of this kind of graphics driver are:

- The Parallax TV driver - (Hydra version: tv_drv_010.spin, Other boards: TV.spin)
    - This uses a tile map, which can also be configured to be used like a screen buffer.
- HEL - the graphics driver on the Hydra Book CD
    - Builds on the Parallax TV driver, and adds sprites. (Up to 5 per line)
- The demo SimpleNTSC
    - Has a flag and some color bars hard-coded

The limitation of this pattern is that there is a very limited amount of time available on the single cog for doing anything other than sending pixel and sync data to the video hardware. Especially if you want to use Hi-Color, high horizontal resolutions, display lots of sprites, or preprocess graphic assets such as change their color depth on the fly. In such situations you can use the Cooperative Rendering Pattern.

A bunch of information about programming the Propeller.

Developer Info:

- PinDefs.spin standard (under development)
- Propeller Font
- Development Board Differences- Xtals and pins
- An answer to the question "How many colors can the Hydra produce?"
- Books, References and Tutorials
- Packaging Propeller Software, about how to distribute the software you have written.
- I've found some Propeller Code and want to use / incorporate it into my project, what do I need to do?
- Programming in C (obsolete info about ICC) needs update>
- Programming in C - Catalina
- Programming in Java
- Programming in Pascal
- Programming in Forth
- Download Protocol
- Converting Text Output Display Type from VGA_text to TV_text and vice versa.
- Managing Concurrency (inter COG communications)
- Large Memory Model

Spin:

- BYTE, LONG, WORD
- Strings
- Cracking Open the Propeller Chip - Decoding the Spin Interpreter
- Reset Sequence - What the Propeller does on power-on or reset
- Integer only GPS navigation

Spin Bytecode:

- Spin Byte Code
- Method Calls
- Referencing Globals

Assembler:

- Assembly Programming
- Assembly, step by step
- How to load a spin-variable into assembly-code (load SPIN-variable from hub to cog within assembly using par and rdlong
- How to store assembly-data into hub ram using wrlong
- propasm - An open source alternative assembler for the Propeller.

- My Assembler Routine Is Doing Something Weird! What's Wrong?
- Things you never wanted to know but were forced to find out
- How is RES different from LONG?
- Large Memory Model
- LAS - Largos Assembler, supports standard PASM and LMM with assembler extensions for easy LMM programming
- BYTE, LONG, WORD
- MATH (binary, BCD, integer, fixed and floating point) on the propeller
- 2's complement create it calculate with it
- FFT in propeller assembler

SphinxOS - A Parallax Propeller Operating System

Original Sphinx web page can be found here: Sphinx Web Page
Original threat at Parallax' forums can be found here: Thread

SphinxOS is an operating system for the Propeller based on the pioneering (and quite amazing) work of mpark: Sphinx (A Spin/PASM compiler that runs of the propeller itself). Built on top of his HAL/BIOS concept, together with new drivers and utilities. Several forum members are working right now on different parts of it (p3/p4 of the above mentioned thread).

The first TODO list as of January 5th 2010:

- **Sphinx OS**
  - Extensions - mpark ?
  - Extensions - Mike Green, OBC, localroger ?
  - Use "-1" instead of "0" for status to allow all 256 character code transfers - cluso (no: add high bit $100)
  - Spin Interpreter to use SRAM - cluso (later)
- **Drivers**
  - SD to use latest fsrw - ?
  - 1-pin video - cluso (wip)
  - 1-pin keyboard - cluso (wip)
  - vga - ?
  - FDX (to substitue for keyboard & video) - cluso (completed)
  - LCD (2x40, 128x64) - Rayman, Peter, Drac?
  - Ethernet - ?
  - USB - ?
  - Others - ?
- **Utilities**
  - File transfers (PC to/from FAT16/32) - already done by mpark
  - Xmodem or Y or Z - OBC, Mike Cook, JamesL, Dracula ?
  - Ed - already done by mpark
  - Preditor - CassLan ?
  - Others - ?
- **Languages**
  - Compiler spin/pasm - already done by mpark
  - PropBasic - Bean ?
  - Catalina - Ross ?
  - FemtoBasic - Mike Green, OBC ?
  - Forth ?
  - Others ?
- **Miscellaneous**
  - ZiCog & CPM - heater, cluso & dracula
  - Sound - Ariba, etc ?

    - Games - baggers, etc ?
- **Missing items...**
    - Editor being useable over a serial connection

Proposed Hub Memory Layout v0.010 1Feb2010 by Cluso99

Spin Byte Code

Spin Byte code was long undocumented. However they have been reverse engineered by Cliffe L. Biffle and Robert Vandiver ("asterick"). The meaning of the operand types can be deduced from the source. The names of the byte codes is unofficial. Even though the source code of the SPIN interpreter is released, there is no official list of opcode names whatsoever.

Opcode structure

Opcodes are divided into two main categories: LOWER and UPPER opcodes. Opcodes $00 to $3F are LOWER opcodes and $40 to $FF are UPPER opcodes. That is, if two highest bits of an opcode are zero, the opcode is a LOWER opcode. Otherwise it is an UPPER opcode.

LOWER opcodes are decoded in the main loop with a jump table. UPPER opcodes are handled in a separate subroutine.

Structure of LOWER opcodes

Following is the structure of opcodes, based on the source code of SPIN interpreter. The names of the fields are not official.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	L	L	B	B	C	NZ

The L and B fields tell to which subroutine (of 16 different) the SPIN interpreter jumps to.
L: LONG. These bits mean from what long the jump address is read. (The jump table is in 4 longs, each long has four one-byte pointers). For UPPER instructions
B: BYTE. Which byte of the long is used as a jump destination?
C: CARRY. Value of the C flag is set to the value of this bit.
NZ: NONZERO. Value of the Z flag is set to the negative value of this bit.

Structure of UPPER opcodes

UPPER opcodes are divided into 3 classes: variable ops, memory ops and math ops. Math ops handle mathematical operations and always operate on the stack. Memory ops (memops) operate on memory, addressed by an operand in the stack. Variable ops (varops) also operate on memory, but an index is embedded in the instruction, it is not read from the stack. They are used to save memory as use of memory op would take an extra long.

For variable ops bit 7 is clear and bit 6 is set, ie. opcodes from $40 to $7F are variable ops. For memory ops, bit 7 is set, but either bit 6 or 5 is clear. So opcodes from $80 to $DF are memory ops. For math ops, bits 7 to 5 are set. Therefore opcodes from $E0 to $FF are math ops.

Struture of Variable ops

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	S/V	X	X	X	O	O

S/V-field: Operate on VAR or STACK? Zero means that the instruction operates on VAR region, one means it operates on local stack (function local variables).
X-field: Offset. These three bits tell which LONG to access. The long offset is added to stack pointer or VAR base pointer.
O-field: Operation field. What to do with the memory location?
00: Read (Push result in the stack)
01: Write (Pop value from the stack)
10: Assignment (effect). In this mode, a second opcode (different from the normal opcodes), called an assignment operator, is executed and its result is stored in the target. These opcodes can be math operators, or for example random number operators, sign-extend operators or decrement/increment operators.
11: Push the address of destination into stack.

Structure of Memory ops

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	S	S	I	B	B	O	O

S-field: Size of address operand. 00: Byte; 01: Word; 10: Long; 11: Illegal (would be a math op). If the I-field is set to 1, the index value is shifted left by S bits.
I-field: Index field. If 1, the target register is determined by adding an index popped from stack to a base address determined by the B-field. If 0, an absolute address is popped from the stack.
B-field: Base mode. If I-bit is 1, this indicates what base address the offset is added to. 00: Base is popped from stack. 01: Base is the object base address 10: VAR base. 11: Stack base.
O-field: See Structure of Variable ops.

Structure of Math ops

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	1	1	X	X	X	X	X

X-field: The 5-bit X-field determines the operation to be executed.

Spin Byte Code Instruction List

The following table is extracted from asterick's GEAR source code, and gives a list of the byte codes.

	Byte Code	Operand Type	**Opcode class**
$00	FRAME_CALL_RETURN	OP_NONE	LOWER
$01	FRAME_CALL_NORETURN	OP_NONE	LOWER
$02	FRAME_CALL_ABORT	OP_NONE	LOWER
$03	FRAME_CALL_TRASH ABORT	OP_NONE	LOWER
$04	BRANCH	OP_SIGNED_OFFSET	LOWER
$05	CALL	OP_BYTE_LITERAL	LOWER
$06	OBJCALL	OP_OBJ_CALL_PAIR	LOWER
$07	OBJCALL_INDEXED	OP_OBJ_CALL_PAIR	LOWER
$08	LOOP_START	OP_SIGNED_OFFSET	LOWER
$09	LOOP_CONTINUE	OP_SIGNED_OFFSET	LOWER
$0a	JUMP_IF_FALSE	OP_SIGNED_OFFSET	LOWER
$0b	JUMP_IF_TRUE	OP_SIGNED_OFFSET	LOWER
$0c	JUMP_FROM_STACK	OP_NONE	LOWER
$0d	COMPARE_CASE	OP_SIGNED_OFFSET	LOWER
$0e	COMPARE_CASE_RANGE	OP_SIGNED_OFFSET	LOWER
$0f	LOOK_ABORT	OP_NONE	LOWER
$10	LOOKUP_COMPARE	OP_NONE	LOWER
$11	LOOKDOWN_COMPARE	OP_NONE	LOWER

$12	LOOKUPRANGE_COMPARE	OP_NONE	LOWER
$13	LOOKDOWNRANGE_COMPARE	OP_NONE	LOWER
$14	QUIT	OP_NONE	LOWER
$15	MARK_INTERPRETED	OP_NONE	LOWER
$16	STRSIZE	OP_NONE	LOWER
$17	STRCOMP	OP_NONE	LOWER
$18	BYTEFILL	OP_NONE	LOWER
$19	WORDFILL	OP_NONE	LOWER
$1a	LONGFILL	OP_NONE	LOWER
$1b	WAITPEQ	OP_NONE	LOWER
$1c	BYTEMOVE	OP_NONE	LOWER
$1d	WORDMOVE	OP_NONE	LOWER
$1e	LONGMOVE	OP_NONE	LOWER
$1f	WAITPNE	OP_NONE	LOWER
$20	CLKSET	OP_NONE	LOWER
$21	COGSTOP	OP_NONE	LOWER
$22	LOCKRET	OP_NONE	LOWER
$23	WAITCNT	OP_NONE	LOWER
$24	READ_INDEXED_SPR	OP_NONE	LOWER
$25	WRITE_INDEXED_SPR	OP_NONE	LOWER
$26	EFFECT_INDEXED_SPR	OP_EFFECT	LOWER
$27	WAITVID	OP_NONE	LOWER

$28	COGINIT_RETURNS	OP_NONE	LOWER
$29	LOCKNEW_RETURNS	OP_NONE	LOWER
$2a	LOCKSET_RETURNS	OP_NONE	LOWER
$2b	LOCKCLR_RETURNS	OP_NONE	LOWER
$2c	COGINIT	OP_NONE	LOWER
$2d	LOCKNEW	OP_NONE	LOWER
$2e	LOCKSET	OP_NONE	LOWER
$2f	LOCKCLR	OP_NONE	LOWER
$30	ABORT	OP_NONE	LOWER
$31	ABORT_WITH_RETUR N	OP_NONE	LOWER
$32	RETURN	OP_NONE	LOWER
$33	POP_RETURN	OP_NONE	LOWER
$34	PUSH_NEG1	OP_NONE	LOWER
$35	PUSH_0	OP_NONE	LOWER
$36	PUSH_1	OP_NONE	LOWER
$37	PUSH_PACKED_LIT	OP_PACKED_LITERAL	LOWER
$38	PUSH_BYTE_LIT	OP_BYTE_LITERAL	LOWER
$39	PUSH_WORD_LIT	OP_WORD_LITERAL,	LOWER
$3a	PUSH_MID_LIT	OP_NEAR_LONG_LITE RAL,	LOWER
$3b	PUSH_LONG_LIT	OP_LONG_LITERAL,	LOWER
$3c	UNKNOWN OP $3C	OP_NONE	LOWER
$3d	INDEXED_MEM_OP	OP_MEMORY_OPCOD E,	LOWER

$3e	INDEXED_RANGE_MEM_OP	OP_MEMORY_OPCODE	LOWER
$3f	MEMORY_OP	OP_MEMORY_OPCODE	LOWER
$40	PUSH_VARMEM_LONG_0	OP_NONE	VAROP
$41	POP_VARMEM_LONG_0	OP_NONE	VAROP
$42	EFFECT_VARMEM_LONG_0	OP_EFFECT	VAROP
$43	REFERENCE_VARMEM_LONG_0	OP_NONE	VAROP
$44	PUSH_VARMEM_LONG_1	OP_NONE	VAROP
$45	POP_VARMEM_LONG_1	OP_NONE	VAROP
$46	EFFECT_VARMEM_LONG_1	OP_EFFECT	VAROP
$47	REFERENCE_VARMEM_LONG_1	OP_NONE	VAROP
$48	PUSH_VARMEM_LONG_2	OP_NONE	VAROP
$49	POP_VARMEM_LONG_2	OP_NONE	VAROP
$4a	EFFECT_VARMEM_LONG_2	OP_EFFECT	VAROP
$4b	REFERENCE_VARMEM_LONG_2	OP_NONE	VAROP
$4c	PUSH_VARMEM_LONG_3	OP_NONE	VAROP
$4d	POP_VARMEM_LONG_	OP_NONE	VAROP

	3		
$4e	EFFECT_VARMEM_LONG_3	OP_EFFECT	VAROP
$4f	REFERENCE_VARMEM_LONG_3	OP_NONE	VAROP
$50	PUSH_VARMEM_LONG_4	OP_NONE	VAROP
$51	POP_VARMEM_LONG_4	OP_NONE	VAROP
$52	EFFECT_VARMEM_LONG_4	OP_EFFECT	VAROP
$53	REFERENCE_VARMEM_LONG_4	OP_NONE	VAROP
$54	PUSH_VARMEM_LONG_5	OP_NONE	VAROP
$55	POP_VARMEM_LONG_5	OP_NONE	VAROP
$56	EFFECT_VARMEM_LONG_5	OP_EFFECT	VAROP
$57	REFERENCE_VARMEM_LONG_5	OP_NONE	VAROP
$58	PUSH_VARMEM_LONG_6	OP_NONE	VAROP
$59	POP_VARMEM_LONG_6	OP_NONE	VAROP
$5a	EFFECT_VARMEM_LONG_6	OP_EFFECT	VAROP
$5b	REFERENCE_VARMEM_LONG_6	OP_NONE	VAROP
$5c	PUSH_VARMEM_LONG_7	OP_NONE	VAROP

$5d	POP_VARMEM_LONG_OP_NONE 7	VAROP
$5e	EFFECT_VARMEM_LO OP_EFFECT NG_7	VAROP
$5f	REFERENCE_VARME OP_NONE M_LONG_7	VAROP
$60	PUSH_LOCALMEM_LOOP_NONE NG_0	VAROP
$61	POP_LOCALMEM_LONOP_NONE G_0	VAROP
$62	EFFECT_LOCALMEM_ OP_EFFECT LONG_0	VAROP
$63	REFERENCE_LOCALM OP_NONE EM_LONG_0	VAROP
$64	PUSH_LOCALMEM_LOOP_NONE NG_1	VAROP
$65	POP_LOCALMEM_LONOP_NONE G_1	VAROP
$66	EFFECT_LOCALMEM_ OP_EFFECT LONG_1	VAROP
$67	REFERENCE_LOCALM OP_NONE EM_LONG_1	VAROP
$68	PUSH_LOCALMEM_LOOP_NONE NG_2	VAROP
$69	POP_LOCALMEM_LONOP_NONE G_2	VAROP
$6a	EFFECT_LOCALMEM_ OP_EFFECT LONG_2	VAROP
$6b	REFERENCE_LOCALM OP_NONE EM_LONG_2	VAROP
$6c	PUSH_LOCALMEM_LOOP_NONE	VAROP

	NG_3	
$6d	POP_LOCALMEM_LONG_3	OP_NONE VAROP
$6e	EFFECT_LOCALMEM_LONG_3	OP_EFFECT VAROP
$6f	REFERENCE_LOCALMEM_LONG_3	OP_NONE VAROP
$70	PUSH_LOCALMEM_LONG_4	OP_NONE VAROP
$71	POP_LOCALMEM_LONG_4	OP_NONE VAROP
$72	EFFECT_LOCALMEM_LONG_4	OP_EFFECT VAROP
$73	REFERENCE_LOCALMEM_LONG_4	OP_NONE VAROP
$74	PUSH_LOCALMEM_LONG_5	OP_NONE VAROP
$75	POP_LOCALMEM_LONG_5	OP_NONE VAROP
$76	EFFECT_LOCALMEM_LONG_5	OP_EFFECT VAROP
$77	REFERENCE_LOCALMEM_LONG_5	OP_NONE VAROP
$78	PUSH_LOCALMEM_LONG_6	OP_NONE VAROP
$79	POP_LOCALMEM_LONG_6	OP_NONE VAROP
$7a	EFFECT_LOCALMEM_LONG_6	OP_EFFECT VAROP
$7b	REFERENCE_LOCALMEM_LONG_6	OP_NONE VAROP

$7c	PUSH_LOCALMEM_LONG_7	OP_NONE	VAROP
$7d	POP_LOCALMEM_LONG_7	OP_NONE	VAROP
$7e	EFFECT_LOCALMEM_LONG_7	OP_EFFECT	VAROP
$7f	REFERENCE_LOCALMEM_LONG_7	OP_NONE	VAROP
$80	PUSH_MAINMEM_BYTE	OP_NONE	MEMOP
$81	POP_MAINMEM_BYTE	OP_NONE	MEMOP
$82	EFFECT_MAINMEM_BYTE	OP_EFFECT	MEMOP
$83	REFERENCE_MAINMEM_BYTE	OP_NONE	MEMOP
$84	PUSH_OBJECTMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$85	POP_OBJECTMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$86	EFFECT_OBJECTMEM_BYTE	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$87	REFERENCE_OBJECTMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$88	PUSH_VARIABLEMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$89	POP_VARIABLEMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$8a	EFFECT_VARIABLEMEM_BYTE	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$8b	REFERENCE_VARIABLEMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP

$8c	PUSH_LOCALMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$8d	POP_LOCALMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$8e	EFFECT_LOCALMEM_BYTE	OP_UNSIGNED_EFFECTED_OFFSET,	MEMOP
$8f	REFERENCE_LOCALMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$90	PUSH_INDEXED_MAINMEM_BYTE	OP_NONE	MEMOP
$91	POP_INDEXED_MAINMEM_BYTE	OP_NONE	MEMOP
$92	EFFECT_INDEXED_MAINMEM_BYTE	OP_EFFECT	MEMOP
$93	REFERENCE_INDEXED_MAINMEM_BYTE	OP_NONE	MEMOP
$94	PUSH_INDEXED_OBJECTMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$95	POP_INDEXED_OBJECTMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$96	EFFECT_INDEXED_OBJECTMEM_BYTE	OP_UNSIGNED_EFFECTED_OFFSET,	MEMOP
$97	REFERENCE_INDEXED_OBJECTMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$98	PUSH_INDEXED_VARIABLEMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$99	POP_INDEXED_VARIABLEMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$9a	EFFECT_INDEXED_VARIABLEMEM_BYTE	OP_UNSIGNED_EFFECTED_OFFSET,	MEMOP
$9b	REFERENCE_INDEXE	OP_UNSIGNED_OFFSE	MEMOP

	D_VARIABLEMEM_BYTE		
$9c	PUSH_INDEXED_LOCALMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$9d	POP_INDEXED_LOCALMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$9e	EFFECT_INDEXED_LOCALMEM_BYTE	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$9f	REFERENCE_INDEXED_LOCALMEM_BYTE	OP_UNSIGNED_OFFSET	MEMOP
$a0	PUSH_MAINMEM_WORD	OP_NONE	MEMOP
$a1	POP_MAINMEM_WORD	OP_NONE	MEMOP
$a2	EFFECT_MAINMEM_WORD	OP_EFFECT	MEMOP
$a3	REFERENCE_MAINMEM_WORD	OP_NONE	MEMOP
$a4	PUSH_OBJECTMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$a5	POP_OBJECTMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$a6	EFFECT_OBJECTMEM_WORD	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$a7	REFERENCE_OBJECTMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$a8	PUSH_VARIABLEMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$a9	POP_VARIABLEMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$aa	EFFECT_VARIABLEM	OP_UNSIGNED_EFFEC	MEMOP

	EM_WORD	TED_OFFSET	
$ab	REFERENCE_VARIABLEMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$ac	PUSH_LOCALMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$ad	POP_LOCALMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$ae	EFFECT_LOCALMEM_WORD	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$af	REFERENCE_LOCALMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$b0	PUSH_INDEXED_MAINMEM_WORD	OP_NONE	MEMOP
$b1	POP_INDEXED_MAINMEM_WORD	OP_NONE	MEMOP
$b2	EFFECT_INDEXED_MAINMEM_WORD	OP_EFFECT	MEMOP
$b3	REFERENCE_INDEXED_MAINMEM_WORD	OP_NONE	MEMOP
$b4	PUSH_INDEXED_OBJECTMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$b5	POP_INDEXED_OBJECTMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$b6	EFFECT_INDEXED_OBJECTMEM_WORD	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$b7	REFERENCE_INDEXED_OBJECTMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$b8	PUSH_INDEXED_VARIABLEMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$b9	POP_INDEXED_VARIA	OP_UNSIGNED_OFFSE	MEMOP

	BLEMEM_WORD	T	
$ba	EFFECT_INDEXED_VARIABLEMEM_WORD	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$bb	REFERENCE_INDEXED_VARIABLEMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$bc	PUSH_INDEXED_LOCALMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$bd	POP_INDEXED_LOCALMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$be	EFFECT_INDEXED_LOCALMEM_WORD	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$bf	REFERENCE_INDEXED_LOCALMEM_WORD	OP_UNSIGNED_OFFSET	MEMOP
$c0	PUSH_MAINMEM_LONG	OP_NONE	MEMOP
$c1	POP_MAINMEM_LONG	OP_NONE	MEMOP
$c2	EFFECT_MAINMEM_LONG	OP_EFFECT	MEMOP
$c3	REFERENCE_MAINMEM_LONG	OP_NONE	MEMOP
$c4	PUSH_OBJECTMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$c5	POP_OBJECTMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$c6	EFFECT_OBJECTMEM_LONG	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$c7	REFERENCE_OBJECTMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$c8	PUSH_VARIABLEMEM	OP_UNSIGNED_OFFSE	MEMOP

	_LONG	T	
$c9	POP_VARIABLEMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$ca	EFFECT_VARIABLEMEM_LONG	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$cb	REFERENCE_VARIABLEMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$cc	PUSH_LOCALMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$cd	POP_LOCALMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$ce	EFFECT_LOCALMEM_LONG	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$cf	REFERENCE_LOCALMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$d0	PUSH_INDEXED_MAINMEM_LONG	OP_NONE	MEMOP
$d1	POP_INDEXED_MAINMEM_LONG	OP_NONE	MEMOP
$d2	EFFECT_INDEXED_MAINMEM_LONG	OP_EFFECT	MEMOP
$d3	REFERENCE_INDEXED_MAINMEM_LONG	OP_NONE	MEMOP
$d4	PUSH_INDEXED_OBJECTMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$d5	POP_INDEXED_OBJECTMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$d6	EFFECT_INDEXED_OBJECTMEM_LONG	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$d7	REFERENCE_INDEXED_OBJECTMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP

$d8	PUSH_INDEXED_VARIABLEMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$d9	POP_INDEXED_VARIABLEMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$da	EFFECT_INDEXED_VARIABLEMEM_LONG	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$db	REFERENCE_INDEXED_VARIABLEMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$dc	PUSH_INDEXED_LOCALMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$dd	POP_INDEXED_LOCALMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$de	EFFECT_INDEXED_LOCALMEM_LONG	OP_UNSIGNED_EFFECTED_OFFSET	MEMOP
$df	REFERENCE_INDEXED_LOCALMEM_LONG	OP_UNSIGNED_OFFSET	MEMOP
$e0	ROTATE_RIGHT	OP_NONE	MATHOP
$e1	ROTATE_LEFT	OP_NONE	MATHOP
$e2	SHIFT_RIGHT	OP_NONE	MATHOP
$e3	SHIFT_LEFT	OP_NONE	MATHOP
$e4	LIMIT_MIN	OP_NONE	MATHOP
$e5	LIMIT_MAX	OP_NONE	MATHOP
$e6	NEGATE	OP_NONE	MATHOP
$e7	COMPLEMENT	OP_NONE	MATHOP
$e8	BIT_AND	OP_NONE	MATHOP
$e9	ABSOLUTE_VALUE	OP_NONE	MATHOP
$ea	BIT_OR	OP_NONE	MATHOP

$eb	BIT_XOR	OP_NONE	MATHOP
$ec	ADD	OP_NONE	MATHOP
$ed	SUBTRACT	OP_NONE	MATHOP
$ee	ARITH_SHIFT_RIGHT	OP_NONE	MATHOP
$ef	BIT_REVERSE	OP_NONE	MATHOP
$f0	LOGICAL_AND	OP_NONE	MATHOP
$f1	ENCODE	OP_NONE	MATHOP
$f2	LOGICAL_OR	OP_NONE	MATHOP
$f3	DECODE	OP_NONE	MATHOP
$f4	MULTIPLY	OP_NONE	MATHOP
$f5	MULTIPLY_HI	OP_NONE	MATHOP
$f6	DIVIDE	OP_NONE	MATHOP
$f7	MODULO	OP_NONE	MATHOP
$f8	SQUARE_ROOT	OP_NONE	MATHOP
$f9	LESS	OP_NONE	MATHOP
$fa	GREATER	OP_NONE	MATHOP
$fb	NOT_EQUAL	OP_NONE	MATHOP
$fc	EQUAL	OP_NONE	MATHOP
$fd	LESS_EQUAL	OP_NONE	MATHOP
$fe	GREATER_EQUAL	OP_NONE	MATHOP
$ff	LOGICAL_NOT	OP_NONE	MATHOP

The SpinStudio Development System

The SpinStudio theory:
The SpinStudio system is the buffet style development system. Start with a Main Board as your plate, and fill it full of good stuff from the robust selection of peripheral modules. Configure your board your way! Decide for yourself which devices you want to interface with your Propeller.
The system is obsolete-proof! As new ideas are developed, a Peripheral module can be easily created and plugged into your existing Main Board. Also, when the Propeller II is released, keep all your existing Peripheral modules and just upgrade your Main Board. Got a great idea that you want to make into a peripheral module yourself? There's the SpinStudio ProtoCard that you can solder together your own module!

SpinStudio Main Boards and peripheral boards are sold as kits, and assembled by the customer before use. Some basic soldering skills and tools are necessary are required. All components are through-hole and spaced for easy soldering.

More About the SpinStudio MainBoard:
The Main Board has the following features.

- On board voltage regulation both 3.3V and 5V to feed both the Propeller and all 4 Peripheral sockets.
- Socket for 40 pin DIP version of the Propeller
- Socket for a 8 pin DIP serial EEPROM
- 5 mhz Crystal
- 4 pin PropPlug connector
- Power Switch
- Reset switch
- 4 connectors for Peripheral Modules each containing the following signals:
    - 3.3V
    - 5V
    - VSS
    - I2c Bus (SDA and SCL)
    - Access to 8 Propeller IO Pins

New ! The SpinStudio Main Board now includes a free Propeller and EEPROM. The EEPROM will be preloaded with Jeff Ledger's PropDOS.

The following parts are required for use and must be purchased separately

- a 7-9 volt wall transformer - with a 2.1mm center positive barrel connector
- PropPlug for Programming
- Various soldering and hand tools needed for assembly

Upon request, an EEPROM loaded with the latest version of Jeff Ledger's PropDOS will be included with any SpinStudio Main Boards purchased. PropDOS will allow you to load and run compiled Binary files from an SD card with the optional SD Card adapter. This will allow you to get up and running with SpinStudio without the purchase of a PropPlug for programming.

Peripheral Modules that are available now:

There are currently 9 different Peripheral Modules available -

- Mouse/Keyboard
- VGA
- Composite Video / Audio
- Parallel LCD
- PropNIC ethernet adapter
- XBee adapter
- SD card adapter
- Input / Output / Servo
- Blank ProtoCard

More to come!

Stand-alone SpinStudio Modules?

Modifications to Parallax's ProtoBoard can be made so that SpinStudio modules will plug right in! The procedure to convert your Proto Board into a SpinStudio clone is detailed in Jeff Ledger's Propeller Cookbook, which you can access in the tutorial section at uController TutorialsThis is the easy way to add features to your Proto Board. Certain SpinStudio modules were designed to be plugged directly into a solderless breadboard too! Any module that can plug into a solderless breadboard can also be interfaced with any other available development board that you may already own, such as Parallax's Demo Board, EasyProp, PropRPM, PropDongle and others. The following chart explains how Modules can be used.

Module Name	Use with SpinStudio?	Use with modified Proto Board?	Use with Solderless BreadBoard?
Mouse/Keyboard	X	X	
VGA	X	X	
Composite Video/Audio	X	X	
Parallel LCD	X	X	
PropNIC Ethernet adapter	X	X	X
XBee Adapter	X	X	
SD Card Adapter	X	X	X

Input/Output/Servo	X	X
Blank ProtoCard	X	X

Where to Purchase SpinStudio systems or components?
SpinStudio can be purchased directly through the uController.com website. Also find useful Reference material and Tutorials on the Tutorials page of the same site.

Or contact Brian Meade directly by any of the following means:

- Email to - Brian@uController.com
- Private message to parts-man73 on Ignite Automation's Forums
- Private message to parts-man73 on Parallax's Forums

Strings

Introduction

A string is just a sequence of characters, one after another, but first; what is a character ?

To humans, characters are simply shapes which we recognise and attribute meaning to singularly and when making up a string or word. A computer or processor like the Propeller has no comprehension or understanding of those shapes. In order to use characters each must be represented in a form which can be used digitally. This was largely done through the American Standard Code for Information Interchange (ASCII) which specified an 8-bit, byte, value which represents the characters we use. Providing everyone agrees on what value a character has we can move characters (and strings) from the digital to our real world and vice-versa. We can deal with characters as shapes, the Propeller can deal with byte values representing those characters.

Digitally then, a string is just a sequence of ASCII codes which represent the values of each character, for example the string "ABC" is represented by three consecutive bytes of hexadecimal value $41, $42 then $43.

String Length

It is convenient to know how long a string is, to know where it ends and where another string (or something else entirely) starts. There are two ways to deal with the length of strings; by prefixing the string with a byte, word or long value which specifies how many bytes there are in the string, or by ending the string with a unique value, much like ending a sentence with a period.

Both have their advantages and disadvantages. A length prefix requires the size of the entity representing the length to be large enough to hold the length of the string or the length of string becomes limited (255 characters for a byte-sized length), but a larger sized entity is wasteful for smaller strings. A mechanism to use variable sized entities depending upon length is possible but makes processing and dealing with strings complicated.

The alternative of using a unique terminating value allows for any arbitrary length of string but the characters of a string must be counted up to the terminating value to determine its length and the terminating value cannot be contained within the string itself.

Strings and Spin

The Propeller Tool chooses to deal with strings in the second way, with a unique terminating value, and this value is chosen to be zero. This is also the way in which the C programming language deals with strings. The common term for such a string representation is "zero terminated string".

The Spin programming language provides three functions which can be used to deal with strings; String, StrSize and StrComp. Any other string processing functions have to be implemented by the Spin programmer themselves.

String

The String directive allocates a sequence of byte values and a zero valued terminator in hub memory and returns a pointer to the first character of the string.

Our previous example, the string "ABC", when created using the String("ABC") function has the following byte value sequence created within hub memory; $41, $42, $43 then $00.

String(..) ist not a function executed during runtime (thus called "directive"). It has an extended syntax in that comma separated values can be used as parameters, concatenated at compile time. Obviously only constants and "literals" can be used for this.
ptr := string("ABC")
is in each and every respect equivalent to
ptr := @_string99
DAT
_string99 LONG BYTE "ABC",0

StrSize

The StrSize function takes a pointer to a string in hub memory, counts how many characters there are up to the zero value terminator and returns that, the size or length of a string.

With StrSize(String("ABC")) the value returned would be 3.

StrComp

The StrComp function takes two pointers each to two strings and compares each byte of the strings and returns a true value (-1) if they are the same byte sequences and a false value (0) otherwise.

With StrComp(String("ABC"),String("ABC")) the value returned would be true, with StrComp(String("ABC"),String("abc")) the value returned would be false. Note that every character has its own unique value so upper and lowercase characters are not the same.

String Handling

Propeller strings (created by the String function) are effectively fixed at compile time and unalterable, read-only. They could be altered but doing so would likely cause incorrect operation of the program and in some cases corruption of the entire program. Read-only strings are useful for displaying and sending messages which do not need to change but strings which are changeable are useful in a number of cases. Changeable strings can be created an manipulated under programmer control.

A string as discussed is simply a sequence of byte values terminated by a zero value. There is no reason that such sequences cannot be created within byte arrays. Once this is done, those byte arrays can be manipulated and used to perform complex string operations.

Setting a String

```
VAR
  byte dstString[256]

PUB Main
  SetString( @dstString, String("ABC") )

PRI SetString( dstStrPtr, srcStrPtr )
  repeat until ( byte[ dstStrPtr++ ] := byte[ srcStrPtr++ ] ) == 0
- or -
PRI SetString( dstStrPtr, srcStrPtr )
  ByteMove(dstStrPtr, srcStrPtr, StrSize(srcStrPtr)+1)
'+1 for zero termination
```

Concatenating Two Strings

```
VAR
  byte dstString[256]
  byte srcString1[256]
  byte srcString2[256]

PUB Main
  SetString( @srcString1, String("ABC") )
  SetString( @srcString2, String("DEF") )
  AddString( @dstSring, @srcString1, @srcString2 )

PRI AddString( dstStrPtr, srcStrPtr1, srcStrPtr2 )
  repeat until ( byte[ dstStrPtr++ ] := byte[ srcStrPtr1++ ] ) == 0
  dstStrPtr--
  repeat until ( byte[ dstStrPtr++ ] := byte[ srcStrPtr2++ ] ) == 0
- or -
PRI AddString( dstStrPtr, srcStrPtr1, srcStrPtr2 ) | len
  len := StrSize(srcStrPtr1)
```

```
  ByteMove(dstStrPtr, srcStrPtr1, len)
  ByteMove(dstStrPtr += len, srcStrPtr2, StrSize(srcStrPtr2)+1)
'+1 for zero termination
```

Appending a Character to a String

```
VAR
  byte dstString[256]

PUB Main
  SetString( @dstString, String("ABC") )
  AppendChar( @dstString2, "D" )

PRI AppendChar( dstStrPtr, char )
  repeat until ( byte[ dstStrPtr++ ] := byte[ srcStrPtr1++ ] ) == 0
  byte[ dstStrPtr-1 ] := char
  byte[ dstStrPtr   ] := 0
- or -
PRI AppendChar( dstStrPtr, char )
  dstStrPtr += StrSize(dstStrPtr)
  byte[ dstStrPtr++ ] := char
  byte[ dstStrPtr   ] := 0
```

Making a String Uppercase

Making a String Lowercase

Getting the Leftmost Characters of a String

Getting the Rightmost Characters of a String

Getting a Sub-String of a String

Finding an Occurrence of a Sub-String Within a String

Other Character and String Representations

A character doesn't have to be a single 8-bit byte. It can be larger (16-bit is often used for unicode) and smaller, either padded to make it a multiple of a common number of bits, or placed bit-contiguous within its storage area. Each character could be of a differing size as it is with morse code where the number of dots and dashes vary according to letter.

A string does not necessarily need a zero terminator; that can be any value which is otherwise unused or defined for the purpose, or a string may begin with a character which is also used to terminate the string, the two not being considered a part of the string itself. The terminator can also be left out entirely where the length of a string is known in advance or it can be indicated by a length which is prefixed before the

string itself.

Strings do not have to be contiguous although they usually are. Non-contiguous strings will require complicated mechanisms to determine where the parts of the the string are which makes processing them difficult.

Working sub- titles: (will publish existence of each in forum when more content for each is ready)

- C multi-tasking - go beyond 8 cogs without thread headaches

- C programs larger than 32K with an hx512

- fishing without a license? ignore your limits!
aka: use an SD card and allow 2gig memory for your C app

- putting events, exceptions and "interrupts" into a C application
... without giving up strict control of timing

- mufflers, meters, valves and jets - pipes on steroids (or sockets)
to couple your cogs

- persistent memory - transparent saving
of hub data back to an EEprom

- persistent memory II - restartable programs and
crash-only software

- persistent memory III - restartable pipes!

- grow your own OS in C, be your own boss!

- C objects that can be "run" by any cog in a storm
... delayed binding and "thunks"

- what good is a bad pointer?

- how real estate issues relate to Prop connectivity
... and how DNS can help point the way to a solution

- the great YAML vs JSON vs serialize() debate...
... and how apps in C on a Prop can win either way

- access to memory in other cogs - even in other Props!

- distributed computing, content addressable cogs

- Subsumption architecture in C on a Prop or three

Obtaining a Cog Address from Spin

To obtain the cog address of a label in a DAT section from spin the method used is -

cogAddress := @ cogLabel >> 2 - @ cogBaseLabel >> 2

This will return the address of the cog label ($000-$1F0).

For example, the cog address of 'CogLabel' is $003 in the following example

```
DAT
              org   $000

CogBaseLabel  nop           ' $000
              nop           ' $001
              nop           ' $002
CogLabel      nop           ' $003
```

This is nice to know, but will there be any use?
Well, an advanced programming style consists of generating your COG machine program "on the fly", or at least thoroughly "parametrize" it using SPIN.
This has some advantages:

- The COG code can become smaller, as it can be computed according to the specificsituation and need not contain all unused variations
- This might be the only way to stay within the COG's 2k limit
- The COG code can become simpler when you (dynamically) arrange all its parameters in a cunning way ("pushing"), rather than let it find everything out itself ("pulling")

Note that the formula above depends on ORG having been set to $000, otherwise this offset has to be added. Note also that using an ORG 0 is VERY advanced :-)

A Thumb-Style VM Implementation for the Propeller Chip

Thumb VM Implementation - Aichip Industries Version - 0.01

All Thumb VM opcodes are 16-bit sized, an 8-bit opcode and an 8-bit operand, or an 8-bit opcode and 24-bit operand for branches and branches to subroutine. The Thumb VM can support up to 32MB of code space.

The Thumb VM provides 256 long registers ($00-$FF) for Thumb VM program use. These are mapped to Cog registers $100-$1FF giving 240 usable registers plus access to the 16 special purpose and hardware registers of the cog. Registers can be used for run-time data or be pre-loaded with long constants.

Most opcodes take the form 'oooooooo rrrrrrrr' where 'o' is an opcode, an 8-bit index into a Cog lookup table ($0xx-$0FF), which specifies the native Propeller opcode to execute, and 'r' is an 8-bit register or immediate value to use with that opcode.

The opcode lookup table contains native Propeller opcodes to use but stored in a modified way to indicate how the 'r' should be used. The lower 8 bits of the source register field are used as the address of a handler within the Thumb VM interpreter to create the correct native opcode to use. Note that this means there must be precise knowledge of the VM layout by the Assembler. It is envisaged that an Assembler would generate a .binary or .eeprom which contains the Spin code to launch the VM, the VM itself plus the VM object code and VM opcode lookup table and register initialisation sequences.

There are a number of types of opcode processing which can be applied -

Source Register
The bottom 8-bits of the native opcode source register are replaced by the 8-bit 'r' value. This facilitates normal 'mov' from source register and immediate loads.
This is used for implementing "mov d,r" and "mov d,#r" VM instructions, where 'd' is defined within the Native Opcode lookup.

Source Indirect
As per Source Register but the 'r' value specifies the register whose contents are used as the effective address of the register used in the native instruction. This indirection avoids the need for self-modifying code.
This is used for implementing "mov d,[r]" VM instructions where 'd' is defined within the Native Opcode lookup.

Destination Register
The bottom 8-bits of the native opcode destination register are moved to the bottom 8-bits of the native source register and the bottom 8-bits of the native opcode destination register are replaced by the 8-bit 'r' value. This facilitates normal 'mov' into destination registers.
This is used for implementing "mov r,s" and "mov r,#k" VM instructions where 's' or 'k' are defined within the Native Opcode lookup.

Destination Indirect
As per Destination Register but the 'r' value specifies the register whose contents are used as the effective address of the register used in the native instruction. This indirection avoids the need for self-modifying code.
This is used for implementing "mov [r],s" and "mov [r],#k" VM instructions where 's' or 'k' are defined within the Native Opcode lookup.

In addition, the opcode types include types for branch (BRA), branch to subroutine (BSR), and return from subroutine (RTS). The actual opcode held in the native opcode lookup table is irrelevant save for the type indicator.

The BRA and BSR VM opcodes have the format 'oooooooo aaaaaaaa aaaaaaaa aaaaaaaa' with the 24-bit address 'a' added to the current pc as a two's complement number. This makes VM code position independent.

The RTS VM opcode has the format 'ooooooo rrrrrrrr'. When the return address is popped the 'r' value is added to it. This allows execution to continue immediately after the BSR or some instruction later. This is convenient for BSR calls which need to behave differently depending upon result -

bsr #TestNumber
bra #LessThanZero
bra #Zero
bra #GreaterThanZero

An Indirect Branch 'bra r' is supported with opcode 'oooooooo rrrrrrrr'. This will cause a branch to the pc which is held within the register 'r'.

The Cog contains a 'hardware stack' to save having to modify hub memory. This is used for subroutine calls but it is possible that VM opcodes can be mapped to native opcodes to manipulate the stack.

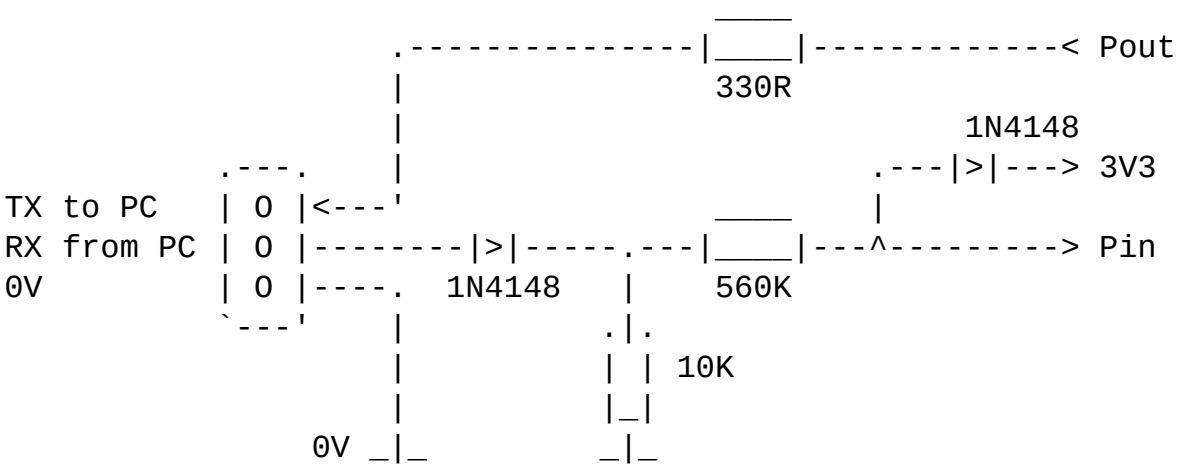
Because BRA, BSR and RTS do not actually use any native opcode, a mechanism has to be found to provide conditional execution. This is achieved by creating a VM opcode which maps to the inverse condition to allow execution and adds 2 or 4 to the PC to skip the following instruction. Such VM opcodes will have the format 'oooooooo 00000010' or 'oooooooo 00000100' and will be mapped to native opcode lookup table type 'source register'. The looked-up native opcode will represent 'IF_X add pc,#r'

Just as it is possible to "Interface 5V signals to the Propeller" higher than 5V signals can be interfaced, providing current limiting is enforced. This can be done using a single resistor.

This can be used to provide a minimal component count, low-cost serial interface for PC or other device connection (for run-time communications, not for program downloading).

A +/-12V RS232 receive line can be connected through just a single resistor and a Propeller output pin can usually drive an RS232 transmit line. This is the absolute minimum circuit and is not recommended. The receive line should be pulled down to prevent the input pin from floating when the serial cable is disconnected, negative input voltages can be blocked by a simple diode, a diode clamp will keep the input pin within accepted voltage range and help protect the internal clamping diodes from any adverse effects, the output pin can be protected by a current limiting resistor.

The author has used the following circuit with no apparent adverse effects for an extended period of time -



The Propeller Chip is conservatively rated to withstand +/-500uA injection current and the 560K resistor ensures that the injection current is considerably below that (+/-21uA at +/-12V).

The circuit can be reduced to just the 560K and 10K, hence the description as a "two resistor interface".

The interface is not RS232 specification compliant and may not be suitable for all circumstances. In particular, the output from the Propeller Chip will be 0V/+3V3 which may be too low for some receiver devices, however this has not been found to be the case with serial interfaces the author has tested.

Link : "Original thread"

Using the Propeller as a USB Host

There is a work-in-progress implementation of a full-speed (12 Mb/s) USB Host now (as of March 2010)

- Source code (subversion repository)
  - http://svn.navi.cx/misc/trunk/propeller/usb-host/
- Forum Discussion
  - http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=440787

Forum Discussions

http://forums.parallax.com/showthread.php?p=760011

Useful Links

Cornell University
http://www.asahi-net.or.jp/~qx5k-iskw/robot/usbhost.html
http://www.mikrocontroller.net/topic/30029

Using the Propeller as a USB Slave

BradC has written a bit-banged USB Slave interface for the Propeller.

This is a run-time, program controlled, interface separate to any USB interface used for downloading into the Propeller and this does not replace nor substitute for a USB2SER or PropPlug.

The USB Slave interface allows the Propeller to connect to a PC host (Windows 98SE, XP etc, Linux and others) using just a standard USB cable.

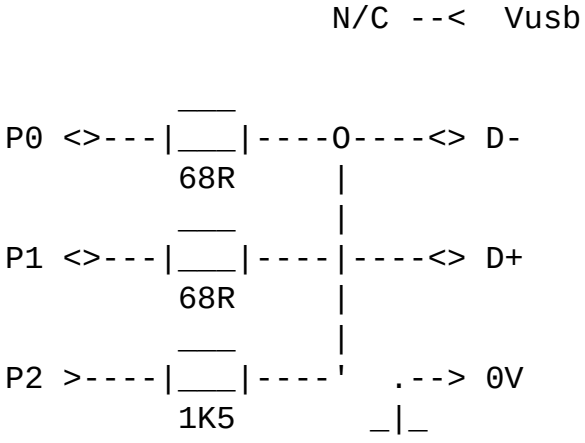
The Propeller interface is a standard USB socket plus three resistors and requires three Propeller I/O lines. Two Propeller Cogs are required to support the USB interface.

When connected and operating, the Propeller will appear as a USB serial device and can be accessed from the PC host through a Virtual Serial Port as if it were just another serial communications port.

BradC has also produced a USB Slave interface which allows the Propeller to emulate a USB keyboard.

More details, including the required Propeller software, Windows Drivers and necessary circuit connection, can be found in the original thread here

USB Hardware Connection



P0 : Data Line Negative (In-Out)
P1 : Data Line Positive (In-Out)
P2 : USB Enable (Out from Propeller)

Propeller I/O pins can be reassigned as required, they do not need to be P0, P1 and P2.

Can be used with Type-A (flat rectangle) or Type-B (square) sockets.

The circuit also appears to work correctly with a 1K2 for P2/D-

Each cog has a video generator module that facilitates transmitting video image data at a constant rate. There are two registers (VCFG and VSCL) and one instruction (WAITVID) which provide control and access to the video generator. The timing signal for the Video Generator is provided by Counter A running in a PLL mode (PLLA). The PLLB of the cog is used to generate the broadcast frequency; whether this is generated depends on if PLLB is running and the values of VMode and VPins.

The Video Generator should be initialized by first starting Counter A, setting the Video Scale Register, setting the Video Configuration Register, then finally providing data via the WAITVID instruction. Failure to properly initialize the Video Generator by first starting PLLA will cause the cog to indefinitely hang when the WAITVID instruction is executed. DIRA must also be correctly set to permit output on the configured pins. While all public registers in a cog are reset when a cog is initialised this doesn't apply to the frame counter. Meaning its initial value depends on chip and cog used, in other words it's **unknown**. Which also means that the initial WAITVID spends *a maximum of 4K* PLLA cycles before it reloads VSCL (the first **known** value, VSCL is loaded when the frame counter reaches zero).

While the Video Generator was created to display video signals, its potential applications are much more diverse. The Composite Video mode can be used to generate phase-shift keying communications of a granularity of 16 or less and the VGA mode can be used to generate any bit pattern with a fully settable and predictable rate.

VCFG – Video Configuration Register
The Video Configuration Register contains the configuration settings of the video generator and consists of several sub-fields:
VCFG[30..29] VMode (video mode) field selects the mode (VGA or composite) and pins used for composite mode

00	Video Generator Disabled, no output
01	VGA mode, 8 bit parallel output on VPins 7:0
10	broadcast on VPins 7:4; baseband on VPins 3:0
11	baseband on VPins 7:4; broadcast on VPins 3:0

VCFG[28] CMode (color mode) selects 0 = two-color mode (pixel data is 32 bits by 1 bit and only colors 0 or 1 are used) or 1 = four-color mode (pixel data is 16 bits by 2 bits, and colors 0 through 3 are used)
VCFG[27] Chroma1 (broadcast color mode) 1 = enables or 0 = disables chroma (color) on the broadcast signal.
VCFG[26] Chroma0 (baseband color mode) 1 = chroma (color) on VPin 0:2 / 4:6 or 0 = chroma (color) on VPin 3 / 7
VCFG[25..23] AuralSub selects COGID of FM audio output for broadcast output on VPins 3 / 7
VCFG[11..9] VGroup selects group of 8 I/O pins used for output (i.e. 3 = P24..P31)
VCFG[7..0] VPins output mask for signals

VSCL – Video Scale Register

The Video Scale Register sets the rate at which video data is generated and has two sub-fields:

VSCL[19.12] PixelClocks the number of PLLA clocks before the next pixel is shifted out by the video generator module. A value of 0 for this field is interpreted as 256.
VSCL[0..11] FrameClocks the number of PLLA clocks that will elapse before the Video Generator Pixel and Color registers and Frame and Pixel counters are reloaded. A value of 0 for this field is interpreted as 4096. It is recommended FrameClocks be an integer multiple of PixelClocks. Since the pixel data is either 16 bits by 2 bits, or 32 bits by 1 bit (meaning 16 pixels wide with 4 colors, or 32 pixels wide with 2 colors, respectively), the FrameClocks is typically 16 or 32 times that of the PixelClocks value.

WAITVID Command/Instruction
The WAITVID instruction is the delivery mechanism for data to the cog's Video Generator hardware. Since the Video Generator works independently from the cog itself, the two must synchronize each time data is needed for the display device. The frequency at which this occurs is dictated by the frequency of PLLA and the FrameClocks field in the Video Scale Register. The cog must have new data available before the moment the Video Generator needs it. The cog uses WAITVID to wait for the right time and then "hand off" this data to the Video Generator. The WAITVID instruction blocks until the Frame counter expires. If the Frame counter expires before the WAITVID instruction blocks then the Video Generator Pixel and Color registers will be loaded with the current contents of the Source and Destination buses, leading to unpredictable output.

The WAITVID instruction passes two longs of data to the Video Generator which are loaded into the Pixel and Color registers. The Colors parameter is a 32-bit value containing four 8-bit color values (although only Color[15..8] and Color[7..0] are used in 2 color mode).

The Pixels parameter describes the pixel pattern to display. The Pixel data is shifted out least significant bits (LSB) first. If CMode = 0 (two color mode), Pixels is a 32x1 bit pattern where each bit specifies which of the two color patterns in the lower 16 bits of Colors should be output to the pins. If the FrameClocks value is greater than 32 times PixelClocks value then the most significant bit is repeated until FrameClocks PLLA cycles have occurred. If CMode = 1 (four color mode), Pixels is a 16x2 bit pattern where each 2-bit pixel is an index into Colors on which data pattern should be presented to the pins. If the FrameClocks value is greater than 16 times the PixelClocks value then the two most significant bits are repeated until FrameClocks PLLA cycles have occurred.

VGA output
For VGA mode, each 8-bit color value is written to the pins specified by the VGroup and VPins field. On the Propeller Demo Board VPin 0 is Horizontal Sync, VPin 1 is Vertical Sync, VPin 2:3 is Blue, VPin 4:5 is Green and VPin 6:7 for 64 color VGA output.

Composite output
For composite video each 8-bit color value is composed of 3 fields. Bits 0-2 are the luminance value of the generated signal. Bit 3 is the modulation bit which dictates whether the chroma information will be generated and bits 4-7 indicate the phase angle of the chroma value. When the modulation bit is set to 0, the chroma information is ignored and only the luminance value is output to pins. When the modulation bit is set to 1 and Chroma0/Chroma1 is 1 then the luminance value is modulated ± 1 with a phase angle

set by bits 4-7. In order to achieve the full resolution of the chroma value, PLLA should be set to 16 times the modulation frequency (in composite video this is called the color-burst frequency).

Broadcast output
Normally, for baseband, the three video resistors form a 3-bit DAC that is 1V-peak under a 75-ohm load. Outputs levels range from #0 to #7 (0 to 1V in ~125mv steps).

For broadcast, those baseband 0-to-7 levels are modulated at the broadcast frequency (CTRB's PLL) in the following pattern:
0 -> 0,7 (bottom of sync = max AC amplitude)
1 -> 1,7
2 -> 1,6
3 -> 2,6
4 -> 2,5
5 -> 3,5
6 -> 3,4
7 -> 4,4 (top of white level = min AC amplitude)
So, this is away to get 8 AC levels from what would otherwise be an 8-level DC DAC.

For aural subcarrier, an extra resistor can be added after relative pins 0-2, on relative pin 3. In the VCFG register, you can select which other COG's CTRA PLL output will be modulated along with the video to provide the aural subcarrier. The Video Generator
XOR's the other COG's CTRA PLL with its own CTRB PLL and outputs that to the fourth pin.
For NTSC, this other-cog's PLL output must be a 100KHz-bandwidth 4.5MHz-center FM signal.

The key to getting good broadcast video is to select a FRQB value that has one's in a span of only 2-3 bits. This keeps the NCO jitter pattern high-frequency enough that the CTRB PLL can filter it out. For example, if you're running the Propeller at 80MHz, you can use $0C00_000 (3/4 of 80 MHz) to generate a very clean 60MHz, which is just below channel 3 in the US. Most TV receiver's PLLs will lock onto this when you go to channel 3.

What good is a *bad pointer*?

what defines *bad* ?

In the bad old days [of C], a bad pointer was one which did not point to
a valid addressable object. This could come from any of 3 things:

1. a dereference on a null pointer - often location 0 in memory had 'code' in it and just as often that code was a valid instruction which when 'de-referenced' yielded either a /good/ (but unexpected) valid memory address or an invalid one. Either way - a bad idea.
2. a non existent memory address - completely outside the physical memory of the machine
3. an unsupported memory address - unsupported may mean valid (present) but likely to be clobbered by the model because its use is reserved to another purpose such as below the stack or amid (freed) heap or code. All but the latter would be subject to change in mysterious ways (stack for instance would vary during an interrupt or call to another function) and the latter (code) had better be the start of a real function. A final 'unsupported' memory address is one which incorrectly points 'inside' an object inadvertently.

Is it possible to turn these 3 cases around to make them either 'safer', useful or meaningful?
I don't plan to make all memory valid - that's what virtualization does... and virtualization /is not/ what we need.
(it really does little to handle case 3).

For case 1 (NULL)

Lets start simple. Can a null pointer be made a little 'safer' in case it IS actually dereferenced. Many C compiler vendors selected one of 3 ways to handle this (sometimes with help of the operating system).

- make low memory non-existent to simply map case #2 back to case #1.
- put the string "Null\0" into the lowest memory location (somehow)
- put a zero at that location so that Null points to Null ad-infinitum (my favorite)

Is this sufficient?
Lets try to see if all of the other cases can be made to work...

For case 3 (unsupported)
Life gets interesting here. When there is plenty of memory and Mhz to go around, one can use 'electric-fence' and similar tools to detect pointers that are given values unexpectedly. On a micro controller like the prop, you don't have this luxury. Never the less, one can at least limit values to be in the ranges of "on the stack"(vs below the stack) "in data" and "in (allocated) heap" and finally "at beginning of function" by doing different kinds of costly runtime checking. We may come back to this one...

For case 2 (non-existent)
On many architectures, a non-existent memory address reference causes a trap or an exception.
This is not true on many newer chips like the Prop.

At a minimum, when one 'walks off the end' of memory some known value is returned (such as all bits on or off). This value can be checked-for though at some performance cost. Perhaps a useful cost if debugging is enabled. Of course if the value IS all off, one can fairly quickly check the condition codes and if the value IS all on, one can compare that to the known highest writable memory address.

[ROM addresses are probably not a concern as very few access to ROM occur in uncontrolled ways.]

The point is, this case often does get caught and mapped into a specific value. It would depend a lot on the application being
able to judge if that value could be confused with with real potential data values (0 or ~0). When following
a linked list, this would be fine as its fair to quickly test for 0 or ~0 at each link juncture.

But what if a non-existent memory address were /intentionally/ used to signal something special?
Remember in the early Apple days when illegal instructions were used to tickle (get into) the operating system?
What if we gave meaning to certain special pointer values?

So long as the pointer wasn't confusable with valid memory locations, then one could use these pointer-values (or ranges thereof) to reference memory on other devices or perhaps even in other systems (given cooperation by the part of the system that 'looks-up' where a pointer goes).

Now hold on before trying to map every device on the planet into a 32 bit address space! I know its tempting for an engineer to want to plan out where everything is. Resist! Such things are the realm of a linker or dynamic library loader (which will be covered in another article) .

Where in the World...

...are Propeller Users?

The Propeller Wiki stats give an indication of where visitors to this site are from are from.
There is a Propeller Head Map where lots of people from the Propeller Chip Forum have marked their location. You can add your own pin if you like.

...is Parallax Inc?

599 Menlo Drive
Rocklin, California 95765
USA

WORD

A word is an unsigned integer. Unlike a long which is signed.

WORD is used as a keyword in 4 different ways:

- In a VAR block
  - `WORD Symbol`
- In a DAT block
  - `WORD ...`
- In a method
  - `WORD [BaseAddressInBytes]`
  - `WORD [BaseAddressInBytes][OffsetInWords]`
- In a method
  - `Symbol.WORD[OffsetInWords]`

WORD Symbol

Declaration of a Spin word variable. Guaranteed to be word aligned. When compiling, Spin groups all the word declarations together in a block after all the long declarations and before the byte declarations, so you can't count on the order of differently sized variables in memory being as in the source. However, all same sized variables will be in the order you declare them.
These variables only exist in Hub memory. They will exist at a place past the binary image combined by PropTool.
They are always initialised to zero.
To access them from assembler, you'd have to pass the address of one to the assembly program through the PAR mechanism and use **RDWORD**/**WRWORD**.

WORD ...

Declare a word aligned label. Size [**WORD**|**LONG**] indicates how much space to allocate for that labelled location. Size defaults to **WORD**. Data will be put into the location modulus the Size field. Layout in memory will reflect the order declared in the source, however differently aligned declarations may result in padding.
The data exists in Hub RAM, and may be copied to Cog RAM when starting a Cog. Spin references will use the original in Hub RAM, Assembler references will use the Cog RAM copy (unless done by reference though **PAR** and **RDWORD**/**WRWORD**).

WORD [BaseAddressInBytes]

WORD [BaseAddressInBytes] [OffsetInWords]

In spin will read/write to a word in Hub RAM. It can only do word aligned read/write, in other words it ignores the least significant bit of BaseAddressInBytes.

```
{{WORD [BaseAddressInBytes] := value}}
'Is equivalent to:
{{BYTE[ BaseAddressInBytes & $FFFE ] := value & $FF}}
{{BYTE[ BaseAddressInBytes | $0001 ] := (value >> 8) & $FF}}

{{WORD [BaseAddressInBytes] [OffsetInWords] := value}}
'Is equivalent to:
{{BYTE[(BaseAddressInBytes&$FFFE)+(OffsetInWords*2)] := value & $FF}}
{{BYTE[(BaseAddressInBytes|$0001)+(OffsetInWords*2)] := (value >> 8) &
 $FF}}
```

Symbol.WORD[OffsetInWords]

In spin will read/write to a word in Hub RAM. Symbol must be a long or a word variable (although as a word, it'd be more straightforward to use simple array indexing - **`Symbol[Offset]`**).

See also

LONG
BYTE
Symbol Address operator