Programming the Parallax Propeller using Machine Language

An intermediate level tutorial by deSilva © 2007
Version 1.21 2007-08-21

Preface

There are permanent requests for guidance to the Propeller Machine Language. Of course everything is well explained in the excellent Parallax documentations; the didactical accent of Parallax however seems to be on how to use SPIN with the hardware features of the Propeller.

But the advanced programmer recognizes soon (it takes ½ hour up to a fortnight) that he has to make his way to machine language programming when he wants to accomplish anything more than blinking LEDs or applying prefabricated "objects".

This tutorial was not written for the beginner: You need already a good understand of the Propeller's architecture and some background from successful SPIN programming. Of course you also know how to work the PropellerTool (=the IDE) and maybe Ariba's most useful PropTerminal.

My intention is not to "start at the very beginning", but to help you over the first frustrations caused by the machine language peculiarities of the Prop.

I only lately discovered that for more than a year now Phil Pilgrim has prepared his "Propeller Tricks and Traps" http://forums.parallax.com/forums/attach.aspx?a=14933 in a quasi complementary way to this tutorial. You will greatly profit from his work after you made it through the first three or four chapters of this tutorial! Some of his "tricks" will surely find their way into my still unwritten "Best Practices" Chapter!

As I have programmed my first micro processor 30 years ago – and that was not the first machine code I got into contact with – I may seem biased and unsympathetic from time to time. Please excuse that! I am open to all suggestions how I can improve this tutorial: Just add a posting or send a PM!

And now: Have Fun!

Hamburg, in August 2007
deSilva

Versions

1.11 Issues wrt layout fixed; starting an Appendix for SPIN
1.20 Major misunderstanding wrt MUX-instruction fixed

Chapter 1: How to start

As the architecture of the propeller differs considerably from other controllers, I shall shortly repeat its main features and components. This is of course well laid down in the Propeller Data Sheet and Manual, and – please do not look too disappointed! – throughout this tutorial I shall present you little more than what you will find in the excellent official documentation. But I shall present it in a different way.

Sidetrack A: What the Propeller is made of

There is 32k ROM, with little interest to us during the first chapters. Plus:

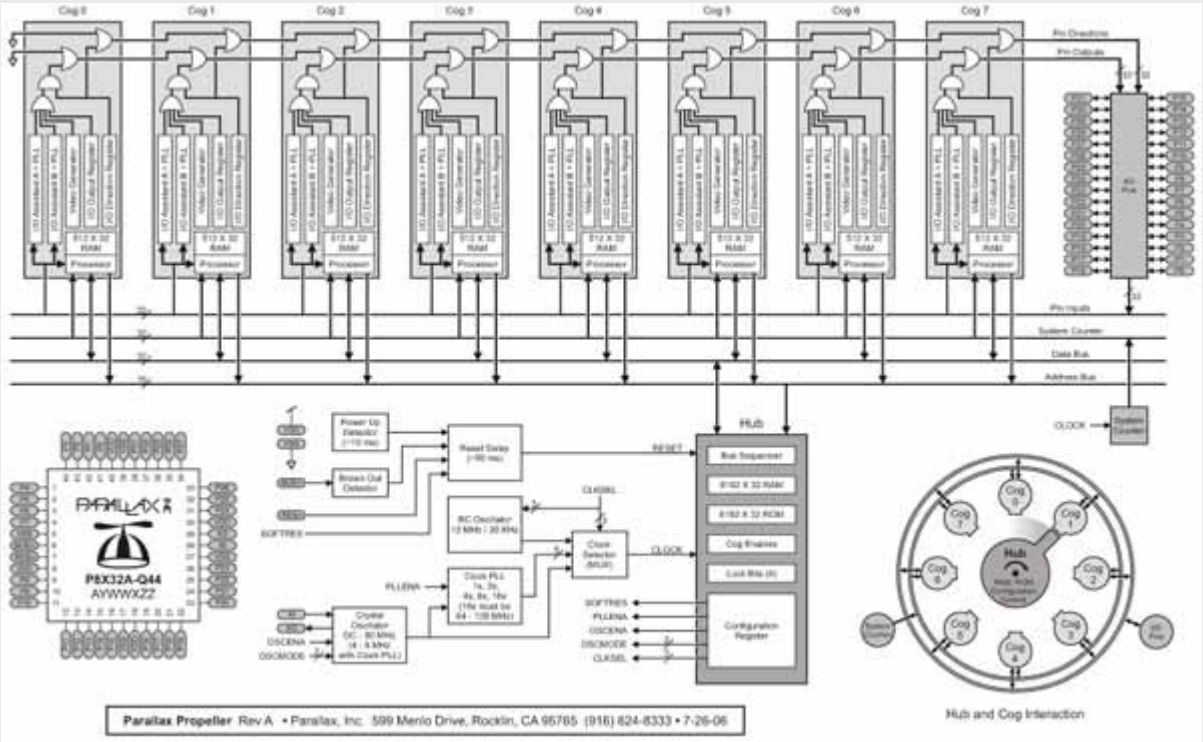
- 32 KB RAM
- 8 processors („COGs“) each running at 20 MIPS
- a 32 Bit I/O Port (“INA, OUTA, DIRA”)
- a system clock („CNT“)
- 8 semaphores (“LOCKs”)

And in each of the 8 COGs:

- 2 KB (512 x 32-bit cells) ultra fast static RAM
- 2 timers/counters (“CFGx”, “PHSx”, “FRQx”, where x = A or B)
- a video processor (“VCFG”, “VSCL”, connected to Timer A)

Note what this adds up to:
160 x 32-bit MIPS - 48 kB static RAM - 16 x 32-bit timers/counters - 8-fold video logic

When you belong to the 75% more visually oriented persons in the world, you may feel more comfortable with the “architectural diagram” of the chip in this Diagram 1



If not already done, take your time to study ALL DETAILS!. (Note: The diagram is web-linked to a hires pdf. Or simply visit the Parallax page!)

When programming in machine language you must generally be very clear with all hardware concepts: The COG-HUB interface, exact timing, working of timer/counters, the "bootstrap". I shall include sections explaining some of those concepts from time to time as "sidetracks"

Sidetrack B: What happens at RESET/Power On?

❶ A part of the ROM is copied into "COG" #0: This is the Bootstrap Loader. It looks at pins 30+31 and tries to serially communicate with the propeller IDE or someone else using the same protocol. (Note: This protocol is open available, but its use is nevertheless a little bit tricky) The data received from the IDE are then stored into HUB-RAM. Optionally they can also be moved into an EEPROM connected at pins 28+29.

However this connection may fail!
In that case:

❷ The lower 32 kB of a serial EEPROM, connected to pins 28+29 are moved into RAM.

If this also fails the Propeller goes idle until the next reset or a new Power-On. Otherwise we now have some defined data in the HUB RAM – copied from the EEPROM or received through the serial connection - that are assumed to be a PROGRAM! Alas, the Propeller cannot execute programs from the HUB-RAM!

❸ During the next bootstrap step, another part of the ROM - the SPIN-Interpreter (Size: 2 KB!) - is copied into processor (=„COG") #0, and – finally! - this program begins – from HUB memory address 16 onwards - to interpret what it assumes is translated SPIN code!

Uff!

Let's talk about processors – called "COGs" in Propeller lingo. What do they do? There is an always correct answer: They execute instructions! A standard processor gets these instructions from a globally addressed memory (in a so called "von-Neumann-architecture") or from a dedicated instruction memory (in a so called "Harvard-architecture" – this is the way PICs and AVRs are organized!). Having two memories allows to "tune" them according to specific needs (e.g. non-volatile, read-only, fast access time), and also to access them in parallel!

A Propeller processor gets its instructions from its internal COG-memory, space limited to 496 instructions! Now, please don't rush to give your Prop to your nephew to play with! Remember, you have 8 of those COGs and the COG-memory is RAM, so it can be reloaded! Furthermore, we have 32-bit instructions, giving them much more power than a common 8-bit instruction has.

So it seems we have a flawless von-Neumann architecture, where instructions and data lay mixed in one memory. Each instruction is 32 bits long and the data – is also 32 bits long. Now this is funny! Does memory not consist of bytes??

No, it does not! It consists of tiny electrical charges caught in semiconductor structures ☺ And it is SOMETIMES packaged in sizes of eight. COG memory is packaged in sizes of 32. Period!

We best call those packages "cells" to avoid misunderstandings! So we have 496 multi-purpose cells, some will contain our program, some data – there are additional 16 cells used as I/O registers; we come to that later.

I know you are now absolutely crazy to have your first instruction executed, but be patient! You have to first learn how your instruction will have its way into a cell of one of the COGs.

Sidetrack C: Loading COGS

We left our last sidetrack with the SPIN interpreter running in COG #0, starting to read things from the HUB-memory. This has to be SPIN byte code, generated by the Propeller IDE, nothing else! So what we need is a SPIN-instruction that will load our bespoken MACHINE-instruction into the "machine", i.e. into an internal cell of a COG. Luckily we already know something like that: It is called COGNEW and it starts a new version of the SPIN Interpreter in a new COG, to interpret a specific SPIN Routine.

Heh, but this is not what we want to do!? Right! But for reasons known only to the inventors loading our own machine code into a COG is also called COGNEW. The first parameter is a HUB address, the second parameter an arbitrary value we can use ad libitum.

```
COGNEW(@myCode,0)
```

This SPIN instructions initiates the copying of nearly 2000 bytes, beginning at *@myCode* into the cells of the next available COG. This is a basic hardware feature of the Propeller (Otherwise, how would it start the bootstrap routine in the first place!), needing no supervision of any kind. One consequence of being such elementary is that it will always load a COG completely, unaware of the meaning or use of those bits it copies...
Note that this will thus always take 500*16/80_000_000 = 100 micro seconds, but the SPIN interpreter will continue his task meanwhile in parallel, performing up to 20 SPIN instructions

Note also that both parameters of COGNEW must be multiples of 4. I know you will forget that immediately, but you have been warned!

I can hear you crying in despair: "BUT WHAT ABOUT MY CODE?" Please! Be patient, we come to that very soon.

The Propeller IDE knows of two different languages: SPIN and Propeller Assembly (or "machine code"). Machine code is encapsulated in the DAT sections, where no SPIN code is allowed. For reasons explained later, we will ALWAYS start our machine code sections with

```
ORG 0
```

and end them with

```
FIT 496
```

Both are NOT machine instructions. They are called assembly directives, and there are very few of them; in fact there is none more but **RES**.

In the DAT section we can use the names of all defined constants or variables of the object as long as it makes sense. We most notably can use the names of all I/O "features" aka I/O registers: **INA, OUTA, DIRA, VCFG, VSCL, PHSA, PHSB, FRQA, FRQB, CFGA, CFGB**.

So let's start!

```
PUB ex01
    cognew(@ex01A, 0)

DAT
        ORG  0
ex01A
        MOV  DIRA,   #$FF   '(Cell 0) Output to I/O 0 to 7
        MOV  pattern, #0    '(Cell 1) Clear a “registers”
loop
        MOV  OUTA, pattern  '(Cell 2) Output the pattern to P0..P7
        ADD  pattern, #1    '(Cell 3) Increment the „register“
        JMP #loop           '(Cell 4) repeat loop

pattern LONG $AAAAAAAA      '(Cell 5)
        FIT 496
```

Before you run this program, make sure you have nothing expensive connected to pins 0 to 7! The Hydra has an LED at pin 0 which will light up and an audio jack at pin 7, which is very convenient.

Before we “look” at the pins using a 'scope or a frequency counter, we do some quick calculations: The (default) RCFAST clock is 12 MHz. With a few notable exceptions each machine instruction takes 4 clocks (Keep this in mind!), so we have 333 ns/instruction: MOV, ADD, JMP. Thus the loop takes **exactly 1 us**. We should now get the following readings:

```
 P0 : 500 kHz
 P1 : 250 kHz
  ...
 P7 : 3.9 kHz
```

Deviations around 3% will be normal with the RC-clock.

This is fast! And imagine, we can run the Prop even 7 times faster!

Now, lets "dissect" our program!
We see some “move-instructions” called **MOV**; it has two “parameters” (or operands). We call the left hand one “dest” and the right hand one “source”. So obviously things are moved from right to left: This is exactly as you write your assignments in SPIN (or in most other languages).

When you have already got experience with a machine language of a common micro processor (8051, 68000, AVR, PIC,..) you will now expect to learn something about “addressing modes” , “registers” etc. etc. You will indeed!

There are two schools of thinking: One (that’s me and the Data Sheet!) says: there are 512 registers in a COG. The other school (that’s the rest of the world) says: There are no registers at all in a COG, except 16 I/O registers memory mapped to addresses 496 till 511.

It is not a problem if you do not follow my way of thinking, you can easily translate it into your own view of the world.

So let's look at the **MOV**-instruction in *Cell 2*: It copies the content of register 5 aka *pattern* into register $1F4 aka OUTA. The **MOV**-instruction in *Cell 0* copies the number 0 into register 5. These are the two addressing modes available in the Prop machine language: register addressing and immediate addressing. (But you will see soon that this is only 97% of the truth: There are some instructions that can move data to and from HUB memory!)

Each and every instruction is able to perform this "immediate addressing" **on its right hand operand**. You indicate this by an "#" symbol in front of this operand, although it is logically a part of the operation code.

What have we else? Ah, there is also an **ADD**-instruction! Obvious what it does: It adds a 1 to register 5.

And nothing more obvious than **JMP**, however … Why do we have this funny "#" here, too?? A typo?

No - think straightforward! When we used *pattern* in the MOV and ADD instruction, we wanted the processor to LOOK INTO that register to load or store that value. When we write #1 (in ADD), we want the processor to use this very value!

So what do we want the processor to do when jumping? NOT look up some register, but just jump to this very cell number we stated: #*loop*.

But! We also could ask the processor to jump to some "computed" destination we stored into a register. This is generally called indirect jumping, is a very important concept, and essential for all subroutine calls.

It is very easy for the beginner to forget the "#", and as this is correct code it will not be detected automatically. If your program terminates in a funny way, first look at all your JMPs for this mistake!

The last line in the program looks familiar: This is just the way we used the DAT section before. Defining and presetting variables. But note that after this DAT section has been copied into a COG (via the COGNEW instruction) the processor looks at it in a different way than the SPIN interpreter does at the "archetype" in the HUB! For the COG it is "register 5"; look for yourself what it can be in HUB: just press F8 and study the memory map. I set pattern to $AAAAAAAA so that you can find it easier.

To finish this first chapter – and before going on to explain more instructions and programming techniques – we shall memorize the structure of the 32 bits of an instruction:

```
6 Bits: instruction or operation code (OPCODE)
3 Bits: setting flags (Z, C) and result
1 Bit:  immediate addressing
4 Bits: execution condition
9 Bits: dest-register
9 Bits: source register or immediate value
```

You by no means shall learn this by heart! It shall rather give you an impression what's all inside a tiny instruction – and what's not, so you can also understand some constraints…

You see that the range of an immediate value is restricted (between 0 and 511). This is no limitation for JUMPs, as this is exactly the size of the whole COG. But if you want to set or add other values you have to preset them into a dedicated cell, as we did in the example (LONG $AAAAAAAA). Funnily, this takes no additional time! You may be accustomed from other processors, that immediate addressing is FAR more efficient than direct addressing. This is not so with the Prop, as direct addressing is just – register addressing!

And don't worry about the things you do not yet understand, enlightenment comes in the next chapters.

Interlude 1: the Syntax of the Propeller Assembly Language

You have swallowed the first machine language program ex01 – have you already digested it? You should have questions, when you had never seen such code before.

The way you write machine language in the form of an assembly program is very similar through all computers, but not equal. There even exists a standard how to write assembly code, where few are aware of and nobody cares for.

The basic principle is to write one instruction per line, elements of this instruction as: labels, opcode, operands, pre and post-fixes are separated either by blanks, tabs or commas. A comma is generally used when the element "left to it" can contain blanks in a natural way, e.g. when writing a constant formula you should like to have this freedom…

You can also define and preset data cells. Generally such presets can be "chained" – separated by commas for the reason stated above. SPIN programmers should be at ease here as everything is exactly as in SPIN.

The same holds for comments.

There is generally something called "directives", which do not lead to code or data but rather tell the assembler to "arrange" things. A typical "directive" would be a constant definition, but this is independently done in the CON section.

"Macro-Assemblers" can have up to a hundred of directives; but there are just three directives for the Propeller:

```
ORG 0   ‘ start over “counting cells” at 0
FIT n   ‘ rise alarm when the recent cell count surpasses n
RES n   ‘ increment the cell count by n without allocating HUB memory
```

Some important rules:

- Use ORG with 0 only
- Don't try to allocate instructions or data after you used RES
- Always finish with FIT 496

If you are one of those single minded technocratic bean counters like me, you might be interested in what is called “syntax” of the assembly language. There is a fine system for 50 years now for such things, called BNF (“Backus-Naur Formalism”).

```
directive     ::= ORG 0 | FIT constant | resDirective
resDirective  ::= [label] RES constant
label         ::= localLabel | globalLabel
localLabel    ::= ":"identifier
globalLabel   ::= identifier
number        ::= decimal | hexadecimal | binary | quaterny
constant      ::= constantName | number | constantFormula
constantName  ::= label | nameFromCON

instruction   ::= [ label ]
                  [ prefix ] opcode [ dest "," ] source [postfix]*
prefix        ::= IF_C | …
opcode        ::= MOV | …
dest          ::= constant
source        ::= [ "#" ]constant
postfix       ::= WZ | WC | NR

dataItem      ::= [ label ] size constant [ "["constant"]" ]
                  [ ","constant [ "["constant"]" ] ]*
size          ::= LONG | WORD | BYTE

program       ::=
                 [ ORG 0 ]
                 [ label | instruction | dataItem ]*
                 [ resDirective ]*
                 [ FIT constant ]
```

Technically speaking, the Prop has a "two address instruction set" sporting a systematic "option for immediate addressing" (0…511). There are no other systematic addressing modes as known from other processors (indexed, pre/post-in/decrementing). Should we need this (and we shall!) , we shall have to "modify" instructions, in that we compute the requested address and "implant it" into an existing instruction. This has been turned down by computer science for decades. The great Edsgar Dijkstra is supposed to have written an article titled "Self modifying code considered harmful", but the manuscript has got lost. The Propeller however cannot live without this; there are even three handy instructions to support this, called MOVI, MOVS and MOVD. We will work through examples in Chapter 5.

Sidetrack D: Who is afraid of OUTA?

Reconsidering our first code example **ex01**: If we understand this COG concept of parallel processors as displayed in **Diagram 1** correctly, there is more going on in the chip than just the 8-bit counter in our COG. All right we have made sure that we can play with pins 0 to 7 but the incrementing also sets higher bits in OUTA……

The rules are:

- A physical pin is **enabled for output**, when the corresponding DIRA bit in **at least one COG** is set to 1.
- A physical pin, enabled for output, is **set to high** when the corresponding OUTA bit in **at least one COG** is set to 1.

Well, nearly...

The correct second part should read:

- A physical pin, enabled for output, is set to "high" when the corresponding OUTA bit in **at least one of the COGS where it is enabled for output** is set to 1.

Which just means everything is as expected. When still in doubt consult the hi-res of Diagram 1; the relevant AND- and OR-gates are drawn in great detail!

So we now can communicate with "outer space" via ear-offending square waves, but how can we get into contact with the "inner space", the fat 32 kB HUB memory? How can we execute instructions from that memory or access data?

Did you listen carefully? The COG-Processors fetches its instructions from COG memory. No exceptions! That means, IF we want to have larger programs than fitting there we shall have to reload them. This is tricky and how to do it efficiently will be part of the "Master Level Tutorial" ☺ You might find some cryptic remarks in the Parallax Forum: Look for "LMM: Large Memory Model".

But reading or writing data to and from HUB is a snap. There is a set of 6 specific instructions for it (BTW: This is labelled to be a "load-store-architecture" in Computer Science lingo):

- WRBYTE und RDBYTE
- WRWORD und RDWORD
- WRLONG und RDLONG

to be used quite straightforward:

```
   RDBYTE cellInCOG, hubAddress
```

But how do we know of any appropriate HUB address? There are two possibilities:
 - 1) You can provide a parameter with COGNEW, which conventionally is a pointer to some HUB memory, perfect to be used with e.g. RDBYTE. This parameter is "automatically" copied into the cell 496 of the loaded COG and can be symbolically referenced by the name **PAR**. (Remember: It has to be a multiple of 4!)

- 2) The second option is more tricky. Remember that the code to be loaded into a COG is always part of the HUB memory first (lets call this the "archetype", as we already did above). So it can be modified by SPIN instructions. (Imagine: It is quite simple to write an Assembler in SPIN!) But in any case we can set some of the DAT variables **before** we load it into its COG.

Confused? Here is the deconfusing example 2:

```
VAR
    LONG aCounter

PUB ex02
    patternaddr := @aCounter
    COGNEW(@ex02A, 0)
    COGNEW(@ex02B, @aCounter)
    REPEAT
       aCounter++

DAT
        ORG    0
ex02A
        MOV    DIRA, #$FF  ' 0..7 for output
:loop
        RDLONG OUTA, patternAddr
        JMP    #:loop
patternAddr
        LONG   0          ' address of a communication variable
                          ' must be stored here before loading COG

        ORG    0
ex02B
        MOV    DIRA, altPins ' 8..15 for output
:loop
        RDLONG r1, PAR
        SHL  r1, #8
        MOV  OUTA, r1
        JMP    #:loop

altPins LONG $FF<<8
r1      LONG 0
```

I thought this was a place as good as any place to introduce some new concepts. Methods (1) and (2) to establish communication should be clear as daylight now. Of course the second COG we activate has to avoid the pins 0 to 7, so we shift its activity

area 8 pins up. Making such small changes can have unforeseen consequences to machine code: It now is no longer possible to move the output pattern directly into OUTA, we have to "shift" it and meet a new instruction for this. In fact there is a complete family of similar instructions, consisting of:

```
SHL : left shift, filling zeroes
SHR : right shift, filling zeroes
SAR : arithmetic right shift, filling bit 31
ROR : rotate right (i.e. bit 0 connected to bit 31)
ROL : rotate left (i.e. bit 0 connected to bit 31)
RCL : rotate with carry left
RCR : rotate with carry right
```

We also have to introduce a new intermediate cell "r1" – this is the typical use of a "register", just needed between two or three instructions and then to be forgotten. It is best to use a "name convention" for such kinds of cells, especially when your programs become larger. "r1" … "r99" or "A"… "Z" - but you should stay consistent through all your programs.

Have you noticed the ":" ? Surely, but what does it mean? In fact "nothing at all". It just helps to be forgotten after its use, so this name can be re-used in another context. This is very handy for the less imaginative who tends to call his labels "lab", "loop", "rep" or such. The scope of these "local" names is from a "global" label to the next "global" label.

Something you will not have noticed immediately is, that the RDLONG takes much more time than 4 clocks.

Sidetrack E: Why the HUB is called the HUB

Trivial as it sounds: Nearly every aspect of the Propeller is displayed in its basic "architectural diagram 1". You see the HUB and the 8 COGs: The HUB turns and meets a COG each 2 clock cycles, adding up to a cycle time of 16 clock ticks until it returns to the same COG.

The science of parallel processing - older than 30 years – has devised a lot of communication schemes between parallel devices: busses, cross bars,… The COG mechanism of synchronized time slots is the most basic (and stable!) one. It resembles a "bus token protocol" in general communication theory. And it is a waste of bandwidth…

But we do not want to criticize, we want to understand. So each 16th clock cycle a COG is able to read or write to the "main memory". Whenever it tries to do this "out of sync", it has to wait. This is why the timing of the RD.., WR.., and other HUB-instructions is so imprecise: they take 7 to 22 clock ticks, depending on where the COG-wheel is, at the time this instruction has been issued inside a COG.

Once you succeed in a memory transfer, you are "synchronized", i.e. you know you will be connected exactly 16 ticks later. That leaves you 2 intermediate instructions only (= 8 ticks plus 7+ ticks) to again read or write to HUB memory…

Well, get out your 'scope again or connect the loudspeakers to pin 7! What has become of our fine 500 kHz? Oh dear! You understand, why?

Of course we can as well write back into the HUB memory, the instruction is WRLONG (or WRWORD, WRBYTE respectively

```
   WRLONG cogCell, hubAddress
```

Note that the operands are in the same order as with RDLONG. Which means the dataflow is now from left to right! This is the only exception in the system and for that reason a common source of confusion.

When in doubt I memorize this: It is possible to write to and read from the first 512 bytes in HUB-memory using "immediate addressing"! Though this is rarely used, it is part of the general "system". And "immediate" values are only allowed for the right hand operand…

Now, back to less arcane stuff! Subroutines are the building blocks of complex programs and nearly as important as JMPs. Of course they are kind of "jumped to", but allow to "return control to the sender".

If you are an experienced assembly programmer I see sparkles in your eyes: PUSHing parameters to the STACK, CALLing routines recursively, POPing all unneeded garbage!

Oh, dear - I am so sorry! None of that – really, nothing at all!

No, the propeller is a true RISC machine: One clock per instruction (well, four to be honest - but that will soon change with Prop II)! And that holds even for CALLs. Do you know how many clock cycles a CALL instruction needs on an AVR mega8? Look it up!

But this is worth thinking about for a moment: How can you realize a subroutine call without a stack?

Here comes the answer:

```
' ex03A
DAT
        ORG    0
ex03A
        MOV    m10par, #30            ' this number 30 ...
        JMPRET times10_ret, #times10 ' ... to be multiplied by 10

' more code of the main program
        .....

' here starts the subroutine
times10
        MOV    m10par2, m10par    ' make a copy
        SHL    m10par2, #2        ' this yields x 4
        ADD    m10par, m10par2    ' plus 1 = 5
        ADD    m10par, m10par     ' times 2 = 10
        JMP    times10_ret        ' indirect jump

times10_ret
        LONG   0

m10par  LONG   0
m10par2 LONG   0
```

The subroutine performs an optimised multiplication by 10; this is straightforward. Note that we need a lot of intermediate registers (m10par, m10par2) as we cannot PUSH or POP anything…

Again we meet a new instruction

```
   JMPRET ret, subr
```

It stores the return address into the lower 9 bits of cell *ret* and jumps to some place called *subr* - if this is a label (which it generally is), don't forget the "#"!

Returning is quite simple; it's just

```
   JMP ret
```

Note that this is an indirect jump, without any "#"!

There are subroutines which have multiple exits, but most have only one. In this situation we can use a clever trick: Instead of exiting by:

```
              JMP times10_ret
times10_ret   LONG 0
```

we simply code:

```
times10_ret   JMP# 0
```

Uff!

Now lets slow down !
How did everything start in the first place?

```
   JMPRET times10_ret , #times10
```

O.k: it jumps to *times10* AFTER it stored the return address into cell *times10_ret* … Not quite: into **THE LOWER 9 BITS** of cell *times10_ret*.

You see: This is the magic of self modifying code! And you also understand that we need the "#" here because we now want no indirection, as the very return address has been already stored into the instruction.

And if you think that is terribly complicated, you are probably right…

You shall have your break now, but before you spend a sleepless night, I have some medicine for you. There is a shortcut for:

```
JMPRET subr_ret, #subr - which is:  CALL #subr
```

And – hurrah! – there is also a shortcut for:

```
JMP# 0 - which is:  RET
```

And if you think this is not medicine but a placebo, you could again be right ☺

Chapter 3: Flags and Conditions

What we need now is a list of instructions so that we can program useful stuff (FFT, 3D graphics, speech recognition, mp3 decoding,...)

After processors became equipped with more than 8 instructions some years ago, there is only one answer to this request: RTFM. What we can do in this tutorial is just point to the most important ones and shed light on some too cloudy things.

The most useful one is arguable the "decrement-and-jump-if-not-zero" (**DJNZ**) instruction. Most things work more or less differently on the Propeller than on other controllers but the DJNZ is a remarkable exceptions: It works EXACTLY as on other processors having this instruction.

- **DNJZ** is a „conditional jump", used for counting loops.
"IF-decisions" are implemented by two other "conditional jumps"
- **TJZ** "Test and Jump if Zero"
- **TJNZ** "Test and Jump if not Zero"

[i]TBD: We'll see an example soon.[/I]

Those three instruction check a register to find out whether it is "empty" or not. All other conditional instructions depend on so called flags, which have to be evaluated by some previous instruction.

"Conditional Instructions?" This will be new to many experienced programmers! There are a few processors that sport a "skip"-instruction. This can be considered as an "instruction prefix", determining whether the instruction in question is to be executed. However this kind of programming technique is generally not taught in courses and the (compound) instructions become quite long.

The Propeller however has included the concept of "conditional execution" into his basic system. 4 bits of each instruction are dedicated to this purpose!

So that we get all the figures right, we should have mentioned that there are in fact two flags only in each processor, called **C** ("Carry") and **Z** ("Zero"). Theses names have historic roots, as arithmetic overflow (C) and "emptiness" (Z) are two important applications for these flags. Most processors have more (some MUCH more) flags, but the Prop has just 2. Which means that there are 16 states (= $2^{(2*2)}$) to be possibly considered as a "condition":

```
No Carry
No Carry and Not Zero
No Carry and Zero
No Carry or Zero
No Carry or Not Zero
Carry
Carry and Not Zero
Carry and Zero
```

```
Carry or Zero
Carry or Not Zero
Not Zero
Zero
Carry == Zero
Carry <> Zero
Never
Always
```

There exist even more mnemonics, as after certain instructions as "compare" (**CMP**) flags are set to reflect the numerical relations
 < , > , => , =< , == , <>
which are straightforward combinations of the "basic flags", but have additional mnemonics. The reader should consult table 5-2 (p. 369) in the Manual for further details.

Let's repeat: each and every instruction can be executed depending on any combination of the two Flags C and Z; this takes neither additional space nor additional time. You can also execute DJNZ or TJZ as conditional instruction. I have never seen this but it could be useful for VERY tricky programs ☺

Before we can make some instructive examples we have to understand how these two flags are set, reset, or left unchanged. Every experienced assembly programmer knows that this can be a nightmare! A small instruction inserted for some reason in a chain of instructions can destroy a clever and efficient algorithm build on a flag. 8080 instructions are extremely clever, in that 8-bit instructions generally influence flags and 16-bit instructions don't – with exceptions…

Now there is good news! You can forget all past problems, as a Propeller instruction will only influence a flag, if you tell it so! This is indicated by some "postfix" notation: you write **WC** ("with carry flag") or **WZ** ("with zero flag") at the end of the instruction.

Now you only have to memorize in what situation an instruction CAN set any flag (if it is allowed to do this). Some basic rules are:

- Moves, arithmetic, and logical instructions change **Z** whether the result is zero or not.
- Arithmetic instructions change **C** according to an "overflow"
- Logical instructions set **C** to form an even parity of all bits.

For the rest of instructions this is more complex :-)

After all these pages of theory we do need an example: We want to count all set bits in a word (Imagine it's the result of shifting-in 32 samples of some signal and we want to estimate its duty cycle)

```
'ex04A
theWord    long $XXXXXXXXXX
counter    long 0
result     long 0

      MOV result, #0       ' will accumulate the number of bits
      MOV counter,  #32    ' we check so many bits
:loop ROL theWord, #1 WC   ' Carry reflects Bit 31, and rotate
left
      IF_C ADD result, #1
      DJNZ counter, #:loop
```

This program has many benefits: *theWord* remains unchanged after the algorithm has completed; it takes a defined and constant time, and the action (ADD #1) can easily be exchanged against something else.

Here is an alternative:

```
'ex04B
theWord    long $XXXXXXXXXX
result     long 0

      MOV result, #0         'will accumulate the number of bits
:loop
      SHR theWord, #1 WC WZ  'Carry reflects bit0, and right shift
                             'Z indicates empty register
      ADDX result, #0
      IF_NZ JMP #:loop
```

This program destroys *theWord*, but works generally faster; we could also get rid of the counter (6 cells against 8 cells in ex04A)

Note there is not really a difference between

```
   ADDX result, #0
```

and

```
   IF_C ADD result, #1
```

You know "Perl"? Right! "There is more than one way to do it" ☺

Chapter 4. Common and not so common instructions

Learning of a new processor, an often heard question is: "What can he do my old processor can't?" (Note: ships are female , computers male.)

(a) "Number crunching" is out. In fact the floating point simulation is not at all bad, but rather below 100 kFLOPS which leaves a broad gap to mathematical co-processors, not to mention SIMD units sported in PC processors since the P3.

(b) Missing also a fixed point multiplication and division instruction, ambitious signal processing is also out, though some audio applications are feasible.

Don't cry! There is a wonderful set of 32 bit instructions for less ambitious but nevertheless high performance computing:

(c) There is 32-bit addition (**ADD**), subtraction (**SUB**) und compare (**CMP**) signed (**...S**) as well as unsigned, even supporting multi precision arithmetic(**...X**)

(d) **MOV, MOVI, MOVS, MOVD, NEG, ABS** and **ABSNEG**; remember we have 2-address instructions throughout, thus NEG and ABS can also make copies in another register; an example:

```
    NEG regA, #1     ' We just set regA to -1; very handy!
```

(e) Logical/bitwise operations (**AND, ANDN, OR, XOR, TEST**)

(f) A complete set of shift instructions, with an **arbitrary** shift value (0..31), given directly ("immediate") or from a register – this is really a high end feature! (**RCL RCR ROL ROR SHR SHL SAR**)

(g) We discussed the jump-instructions already in chapter 3:

TJZ r, jumpdest
TJNZ r, jumpdest
Jumps, if r is empty, or not empty respectively; note that no "flags" are used or changed

DJNZ r, dest
Similar to TJNZ, but register r is decremented before the test. These three instructions take 4 clocks only , IF they jump. If they „fall through", the instruction pipeline has to start over which needs 4 additional clocks to get in „phase" again.

(h) But the Prop also sports a set of instructions rarely found in other processors. Writing your first machine programs you better avoid them, as misunderstandings might fool you into errors difficult to identify. However there are good reasons to have these instruction, as they speed up certain classes of algorithms considerably!

MAX a, clipval
The main problem with this instruction is that it is the mathematical MINIMUM-operation. The best description what is does is: "upper clipping": it clips "a" to "clipval" if necessary.

This is a heavily used operation in all kinds of graphics programming.

MIN a, clipval
This is „lower clipping", promoting „a" to „clipval", had it been less before (i.e. the mathematical MAXIMUM operation). So be careful for the funny name it has been given!

Both instructions are also available as a signed variant **(MINS, MAXS)**

CMPSUB a,b
Subtracts „b" from „a", but only if this should leave a non-negative value in "a". This will support division; here a most trivial application:

```
' ex07A
' compute  c := a divided by b; c and b assumed to be positive
     MOV     c, #0
:loop
     CMPSUB  a,  b WC WZ 'Carry is set, if operation performed
     IF_C          ADD  c, #1
     IF_C_AND_NZ   JMP  #:loop
' the division remainder is in "a"  now
```

We also could have coded (without any further advantage):

```
' ex07B
' compute  c := a divided by b; a and b assumed to be positive
      NEG     c, #1
:loop
      ADD  c,  #1
      CMPSUB  a,  b WC WZ ' Carry is set, if operation performed
      IF_C_AND_NZ   JMP  #:loop
' the division remainder is in "a" now
```

We shall see an advanced version of division later! Can you already imagine, how it will differ? Hint: Think what you have learnt in school ☺

(i) A very peculiar instruction:
REV a, n
Clears upper n Bits und reverses the sequence of the (32-n) lower bits of register "a".
Reverse? That is: bit0 <-> bit 32-n, bit1 <-> bit 32-n-1, etc.

(j) And there also is a somewhat isolated very special subtraction instruction
SUBABS a,b doing a := a - |b|

(k) The next instructions come in groups of 4, as their results depends on the setting of one of the flags, either C, NC, Z, or NZ

Four „multiplex" instructions
MUX* r, *mask*
Set all bits in register r with a corresponding ONE in *mask* to ONE or ZERO, depending on the following table. Sorry, I tried to explain in other ways but it either did not work or had been wrong

☺ Bits in r where the corresponding bit in *mask* is ZERO are not affected at all.

```
  |C |NC|Z |NZ  <- *
--+------------
C | 1  0  -  -
NC| 0  1  -  -
Z | -  -  1  0
NZ| -  -  0  1
 ^
 |
Flag
```

We have also 4 similar conditional „negate" instructions
   NEG* dest, source
depending on the flags C, NC, Z or NZ either a **MOV** or a **NEG** is executed. Equivalent code would be:

```
IF_*  NEG dest, source
IF_N* MOV dest, source
```

Last and least we have 4 „account balancing" instructions
   SUM* sum, source
depending on the flags C, NC, Z or NZ either an **ADDS** or a **SUBS** is executed.

(l) The last group of instructions is well known from SPIN programming:

CLKSET
COGID COGINIT COGSTOP
LOCKNEW LOCKRET LOCKCLR LOCKSET
WAITCNT

They work similar as their SPIN-equivalents, so there is no need to elaborate on them further (You do know SPIN, don't you?)

Interlude 2: Some arithmetic examples

I think it is now time to present you some professional code you should be able to understand by now, taken from the Parallax libraries: multiplication, division, and square roots. I left the original comments.

```
' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
mult  shl   x,#16     'get multiplicand into x[31..16]
      mov   t,#16     'ready for 16 multiplier bits
      shr   y,#1 wc   'get initial multiplier bit into c
:loop
      if_c add y,x wc 'conditionally add multiplicand into product
      rcr   y,#1 wc   'get next multiplier bit into c.
                      ' while shift product
      djnz  t,#:loop  'loop until done

mult_ret
      ret             'return with product in y[31..0]
```

This is shorter than you thought, isn't it? Just 7 instructions! But time consuming! And it's not a 32x32 multiplication but 16x16. This is very common; the hardware multiplication in the Prop II will most likely also be a 16x16 multiplication only. But you can easily build up on it to 32x32. How? Well, remember “binoms” from shool? No?

It's: $(a+b)^2 = a^2 + 2*a*b + b^2$
The rest is simple coding…

The algorithm should be clear to you: It is “standard” multiplication in the same way you do it in the decimal system. However it has one great advantage: The multiplication table is just 1x1=1 ☺

When you do not understand a program you must “trace” it, step by step: Make a list with columns for all relevant variables, and write down - line for line - how they change!

```
' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]
'
divide   shl y,#15     'get divisor into y[30..15]
         mov t,#16     'ready for 16 quotient bits
:loop    cmpsub x,y wc 'if y =< x then subtract it, set C
         rcl x,#1      'rotate c into quotient, shift dividend
         djnz t,#:loop 'loop until done
divide_ret
         ret           'quotient in x[15..0], rem. in x[31..16]
```

This should be also much shorter than you expected! (6 instructions). Note the clever use of CMPSUB! As we are dealing in the binary system, there is no need to “loop” CMPSUB as we did in our preliminary example ex07 above! The algorithm is “school division” - in the binary system.

```
' Compute square-root of y[31..0] into x[15..0]
'
root          mov     a, #0        'reset accumulator
              mov     x, #0        'reset root
              mov     t, #16       'ready for 16 root bits

:loop         shl     y, #1   wc   'rotate top two bits of y …
              rcl     a, #1        ' … into accumulator
              shl     y, #1   wc
              rcl     a, #1
              shl     x, #2        'determine next bit of root
              or      x, #1
              cmpsub  a, x    wc
              shr     x, #2
              rcl     x, #1
              djnz    t, #:loop    'loop until done

root_ret      ret                  'square root in x[15..0]
```

This is left for your own ingenuity ☺

Chapter 5: Indirect and indexed addressing

Indexed addressing is needed when we want to extract some element from a "vector": X [I] . (Indirect addressing is a special case with I == 0.) Other processors use different concepts to accomplish this need, sometimes limiting either "index" or "base address" to eight or sixteen bits… Post- and pre-incrementing the index is also a common option. To close this discussion, there is nothing at all of this kind within the Prop ☺

Of course we have learnt to access the HUB memory using RDLONG or WRLONG with a pre-computed address in a cog register. It might look like this:

```
'ex07A
   MOV r, I
   SHL r, #2          ' x 4 = byte address in HUB
   ADD r, X
   RDLONG r, r
```

To become familiar with this kind of memory access let's just compute the sum of 20 numbers allocated in HUB memory:

```
'ex07B
   MOV addr, X
   MOV sum, #0
   MOV count, #20
:loop
   RDLONG r, addr
   ADDS sum, r
   ADD addr, #4     ' the next long
   DJNZ count, #:loop
```

But how do we manage things when this 20-number-vector is allocated inside our own COG?

Enter self modifying code!

```
'ex07C
' we assume X to X+19 contain 20 longs to be added up
     MOVS :access, #X  ' this instruction modifies a COG cell (*)
     MOV sum, #0
     MOV count, #20
:loop
:access
     ADDS sum, 0-0     ' the lower 9 bits of this instruction…
                       ' … will be modified by (*)
     ADD :access, #1   ' modify a cell to point to the next number
     DJNZ count, #:loop
     …
X:   RES 20
```

The alert reader has spotted a new instruction : **MOVS** ! What's that? Well, there are three specific MOV-instruction taking into account the need for modifying instructions; **MOVS** ("source") will only store to bits 0 to 8 ; **MOVD** ("dest") will only store to bits 9 to 17, and MOVI ("instruction") will only store to bits 23 to

31. Forget to use these instructions for clever byte manipulation; they are meant for 9-bit manipulation :-) But some I/O registered are organized in a way you can utilize these instructions.

And note: There is a strict rule: NEVER modify the next instruction to be executed! We shall elaborate on this in Sidetrack F!

I said: "…not meant for byte manipulation…", but with a little help from Fred Hawkins I devised this little gem:

```
'ex08A
'How to pack 4 bytes into a COG cell and write it to the HUB

  MOVI x, byte0  ' store to the upper 9 bits…
                 ' … leaving bit 31 a "don't care"
  SHR x, #8      ' shift right so upper 9 bits become free again
  MOVI  x, byte1 ' repeat…
  SHR x, #8
  MOVI  x, byte2
  SHR x, #7      ' Attention! Don’t shift out the LSB …
  SHR x, #1 WC   ' … but keep it in the carry flag
  MOVI  x, byte3
  RCL x, #1      ' shift LSB back: bit 31 was a don’t care …
                 ' …  but now no longer is
  WRLONG x, huba
```

It can be done much simpler by this code:

```
'How to pack 4 bytes into a COG cell and write it to the HUB
'ex08B
  WRBYTE x, byte0
  ADD  x, #1
  WRBYTE x, byte1
  ADD  x, #1
  WRBYTE x, byte2
  ADD  x, #1
  WRBYTE x, byte3
```

But how much time will ex08B take? Compare it to ex08A!
Now, but wait! What about the "order of the bytes"? In example A byte0 was the LSB and now … It’s the MSB!!

Well, not really! This has to do with something already Capt’n Gulliver had his issues with: Little Endians! A LONG in the Propeller HUB is stored in a way that its LSB comes first, and the MSB last. No further comment, except you will never notice this inside the COG, as you cannot break-up its internal cell structure.

Can we "improve" ex08B further?
Yes, we can! Assume *byte0* to *byte3* are contained in consecutive registers as follows:

```
byte0 long “a”
byte1 long “b”
byte2 long “c”
byte3 long “d”
```

```
'How to pack 4 bytes into a COG cell and write it to the HUB
'ex08C
   MOVD wrpatch:, #byte0 ' “implant” byte0's addr into instruction
   MOV count, #4         ' prepare to store 4 bytes in the loop
:wrloop
:wrpatch
   WRBYTE 0-0, byte0     ' the 9 dest-bits 0-0 will be patched
   ADD  :wrpatch, aOneInDestPosition ' a very clever patch
   DJNZ count, #:wrloop


aOneInDestPosition LONG  %1_0_0000_0000
count  LONG 0
```

Doesn’t this look much more fancy! And – listen – it will not take much more time than the ex08B, as we had some "time to spend" between the WRs…

Note how we cleverly avoided to hurt the basis rule of Propeller Code Patching: **“Never change the NEXT instruction!”**

And it does not even need more cells (=7) than ex08B ☺

Sidetrack F: How the instruction pipeline works

So you have learnt that most Propeller instructions take 4 ticks (which is 50 ns @ 80 MHz); even I told you so. Well, I am sorry: That was a lie!

A “standard” instruction takes 6 ticks:

```
T=0:  Fetch Instruction
T=1:  Decode instruction
T=2:  Fetch „dest“ operand
T=3:  Fetch „source“ operand
T=4:  Perform operation
T=5:  Store result back into “dest”
```

You can see that most of the time passes in accessing the COG memory (T=0,2,3,5). One could think, it should be nice to skip some of those time slots, if there is no source operand to fetch, because we have an “immediate” instruction (T=3); or T=5, if we do not store back anything.

But the Propeller – like most other advanced processors – uses a completely different approach to speed things up: it “interleaves” memory accesses for the NEXT instruction in the gaps (T=1 and 4), where the memory is not used for the CURRENT instruction. This will look like this:

```
     CURRENT instruction          LAST/NEXT instruction
T=-1:                             Fetch "source" for LAST operation
T=0: Fetch Instruction            Perform LAST operation
T=1: Decode instruction           Store result back into "dest"
T=2: Fetch „dest“ operand
T=3: Fetch „source“ operand
T=4: Perform operation            Fetch NEXT instruction
T=5: Store result into "dest"     Decode NEXT instruction
T=6:                              Fetch "dest" for NEXT instruction
```

There is no way to do it better ! The memory is now used in every cycle; and a new instruction is fetched every 4th cycle.

You can now understand, why it will not work to patch the NEXT instruction, as this is fetched at T=4, whereas the patch only happens at T=5!

And it is important to uphold this inter-locking!

However there are two kinds of instructions that cannot be “locked-in”, as they will take an unknown amount of time. One is the WAIT-familie, the timing of which is something like this:

```
T=0:   Fetch WAIT instruction
T=1:   Decode instruction
T=2:   Fetch "dest" operand
T=3:   Fetch "source" operand
T=3+N: Wait zero to N ticks
T=4+N: Store result back into "dest"
```

Without any wait, this will take 5 ticks. There is no interleave of the NEXT instruction; the fetch of the NEXT instruction will be performed only at T=5+N

Now wait! A "standard" instruction takes 6 ticks, a "waitless wait" just 5; shouldn't it then take just 3 ticks in the context of the pipe flow?

Very clever! But being so "variable" WAIT (and a HUB instruction) is not locked in the pipeline! The fetch of the next instruction – which happens at T = 4 for a standard instruction - does not happen at T = 3+N but at T = 5+N only – or so it seems...

The other exception is the HUB-family. The timing of an RDLONG is something like this:

```
T=0:   Fetch HUB-instruction
T=1:   Decode instruction
T=2:   Fetch "dest" operand
T=3:   Fetch "source" operand
T=3+N: Wait zero to 15 ticks for HUB to sync
T=4+N: Address HUB
T=5+N: LOAD/STORE to/from HUB
T=6+N: Store result back into “dest”
```

We can now also try to understand the timing of a conditional jump instructions; The processor always "predicts" a jump will be taken and fetches the NEXT instruction from this address at T = 4. If the instruction "falls through", this was a bad prediction, and the fetch has to be repeated at T = 6. However T = 6 is not meant to be a "fetch" phase, as the jumps are locked into the pipe, in contrast to the WAIT and HUB instructions. And the next "scheduled fetch" is T=8 ...

Note: The mechanism presented here is not well described in official Parallax documents and depends mostly on my own “educated guesses”.

BTW: Have you also spotted “RES”? In ex07C? This is the last “assembly directive”! It “reserves” memory without allocating it, which will say: There is no HUB memory for these cells! When you think about this for a while you will come to the conclusion, that this is either not possible, or only at the end of the program with no presets or instructions following. And right you are!

Chapter 6 Locks and Semaphores

There is much uneasiness about this: Do I really need "locks"? Or "flags"? But Why? I never needed them before!

Fact is that every parallel system – be it true hardware or faked software – mandatorily needs them, nota bene not for all applications. Thus (binary) semaphores have to exist in the Prop hardware, called "locks".

Consider the following scenario:

> A department store has one entrance and one exit only to better control the stream of daily customers. Now the management wants to learn how many customers (or employees) are in the building at a given time of the day. They think they can optimise staff assignment and close the store more confidently in the evening… High reliable photoelectric relays are installed at the exit and the entrance…

Generally there are two kind of answers the question: „How many persons are inside the store?"

```
ex09A
(- at he entrance-) IF signal THEN inCount += 1
(- at the exit -)   IF signal THEN outCount += 1
(- at the office -) personsInStore := inCount - outCount
```

This approach has some disadvantages:
- both *counts* can overflow
- the result must always be "calculated", thus it is not available "truly" online.

But if both is no issue you should always prefer that solution!

```
ex09B
(- at he entrance-) IF signal THEN personsInStore += 1
(- at the exit -)   IF signal THEN personsInStore -= 1
(- at the office -) display (personssInStore)
```

This is probably the simple solution that comes to mind first, but there is a pitfall here!

If the code – as given – is NOT serially executed, but we have independent processing units near both gates, it can happen that both access the "accumulator" *personsInStore* contemporaneously.

Of course there is no such thing as contemporaneousness, but you never know and this is in fact the basic philosophical issue of digitising signals…

Now assume the department store technician is a Propeller fan ☺ and runs "exit-supervision" in COGA and "entrance-supervision" in COGB

```
' department store ex09C
CON
  entPin = 2
  exPin  = 3
VAR
LONG personsInStore

PUB  main
  PersonsInStore := 0
  cognew(@entrance, @ personsInStore)
  cognew(@exit, @ personsInStore)

DAT
  ORG 0
entrance
  WAITPEQ :null, #entPin
  WAITPNE :null, #entPin
  RDLONG :e, PAR
  ADD :e, #1
  WRLONG :e, PAR
  JMP #entrance

:null LONG 0
:e    RES  1

  ORG 0        'note that this ORG is most important!
exit
  WAITPEQ :null, #exPin
  WAITPNE :null, #exPin
  RDLONG :a, PAR
  SUB :a, #1
  WRLONG :a, PAR
  JMP #exit

:null LONG 0
:a    RES  1
```

It is obvious that you will get into trouble if we encounter the following scenario:

```
RDLONG a,..
ADD a, #1
RDLONG e,..
ADD e, #1
WRLONG a,..
WRLONG e,..
```

One entering customer will not be counted. The probability for this is unknown…

But can we fix this bug principally?

Well, we just have to chain the three instructions: RDLONG - ADD - WRLONG to an unbreakable ("un-interruptable") unit!

Standard processors generally give you one or the other (or both) of the following solutions::
- „Disable Interrupts"

- A special "ReadAndModify" instruction

Having separate hardware, only the second solution is applicable; the „ReadAnd Modify“ instruction on the Propeller is called LOCKSET or LOCKCLEAR respectively.

However it cannot use an arbitrary HUB-cell but one of eight specific bits (called LOCKs) only.

So we can secure our code in the following way:

```
'ex09D

' we enter here when relay issued signal
:l
   LOCKSET  sema  WC
   IF_C JMP #:l  'wait for our partner leaving his "critical area"

' now WE enter our “critical area”
  RDLONG :a, PAR
  SUB :a, #1
  WRLONG :a, PAR
  LOCKCLEAR sema  ' we leave our "critical area"
```

The complete changed program now looks this way; we also “improved” it a little bit in some details the interested reader might profit from understanding…

```
' department store ex09E
VAR
  LONG personsInStore

PUB  main

 PersonsInStore := 0
 semaNumber := LOCKNEW ' rserves a new semaphore
                       ' (Good code would check for < 0)
 countAddress := @ personsInStore

 pin := 2
 delta := 1
 COGNEW(@guard, 0)    ' Entrance
 WAITCNT(cnt+512*16)  ' it takes time to load a COG

 pin := 3
 delta := -1
 COGNEW(@guard, 0)    ' Exit

DAT
guard
  WAITPEQ :null, pin
  WAITPNE :null, pin
:w
  LOCKSET semaNumber WC
  IF_C JMP :w
  RDLONG a, countAddress
  ADD a, delta
  WRLONG a, countAddress
  LOCKCLR semaNummer

' debounce switch
  MOV a, :debounceTime
  ADD a, CNT
  WAITCNT a, #0
  JMP #guard

:null long 0
:debounceTime  LONG 80_000_000/1000 * 10  ' 10 ms
pin            LONG 0
delta          LONG 0
semaNumber     LONG 0
countAddress   LONG 0
a              RES  1
```

Chapter 7: Video without Video

Another fascinating feature of the Prop is the easiness with which it can generate video signals. There is a little bit magic in the NTSC colour generation, but not much. We shall understand everything after we have worked ourselves through the following three chapters. A little bit lengthy – may be – but not really complex.

There is some specific hardware we shall call video logic in each COG, which is also handy for
- general 8-bit output
- especially if connected to a D/A converter of the R2R kind

This should be no surprise, as video is just an analogue signal as any other (or even three in the case of VGA: R, G, B)

A COG uses one of his timers ("A") to produce a certain clock to drive this video logic. This timer can be programmed in a wide range, thus allowing a wide range of applications! How to do this is explained in Parallax's excellent Application Note AN001. I have no intention to repeat the contents of it here. I do it for the German readers, but it would be folly to retranslate this! Just BTW – you English readers have no idea how privileged you are to understand the excellent Parallax documentation written in your mother language. Do use this privilege!

But we will see the timer working later in the examples of course.

Back to the "video logic": it has three modes of operation
- Composite Video (Baseband)
- Broadband Video ("TV")
- VGA

There are a few peculiarities in the former two that might confuse us in the beginning, so we start with the VGA mode. In this chapter we shall not yet wonder why it is called VGA – it's just a name!

So we set this mode – "Vmode = 01" - in **VCFG** ("Video configuration") - one of the two visible video registers; the other is called **VSCL**- "video scale". You will most likely have to look at the tables in the Propeller Datasheet (or manual) from time to time to find your way through the different fields in the 32-bit configuration register. It makes no sense to copy these tables here.

Whatever mode we have selected, there is not much difference for the rest of the handling.

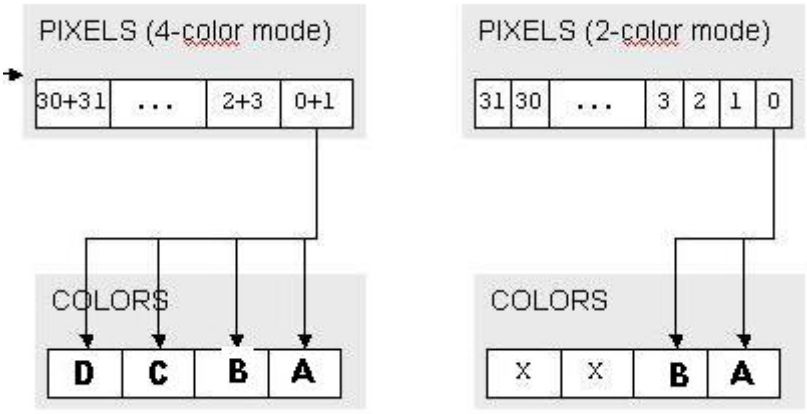
There are also two hidden registers, I like to call PIXELS and COLORS (as I know of no official names): they must exist somewhere in the COG, though they cannot be read but just be set using the WAITVID instruction. This instruction in fact is the theme and centre of this chapter!

Let's imagine we have somehow connected the bespoken clock to the PIXELS register. PIXELS is a (arithmetic) "right-shift-register",

and we are now shifting-out pixels, starting at the LSB end, either bit by bit, [b]or[/b] bit-pairs by bit-pairs. These are two different sub-modes called "2-color-mode" or "4-color-mode" for reasons that will become obvious later.

Now, what happens to these 32 out-shifted bits (or 16 out-shifted bit-pairs)? Where are they shifted to? Good question! To our great surprise, they are NOT shifted out of any I/O pin!

Rather, they are used to [b]address[/b] one of two bytes A or B (or one of four bytes A, B, C, or D) of the COLORS-register! Have a look at this Diagram 2:



Now, exactly one of these bytes A, B, C, or D is output at some configurable 8 pin group (I/O 0..7, 8..15, 16..23) Be careful with group 24..31 the use of which is also possible.

Well, this is nearly all to be said about Propeller video, except some minor details.

Did I already explain how to **set** PIXELS and COLORS? It is done by the WAITVID instruction. So if you want to output a specific bit pattern, just do:

```
WAITVID (eightbitPattern, 0)
                ^           ^
                |           |
            COLORS      PIXELS
```

Now 32 „times" (the most right eight bits of) eightbitPattern is output. When you have selected the 4-color mode this happens only 16 "times"; but it will be difficult to spot the difference…

We call this sequence of output patterns a "register frame" (being of length 32 or 16)

Now we want to do something more adventurous: output a sequence of bytes like $FF, $1F, $07, $00; when we connect a R2R network to the eight pins it would look like a saw tooth at the end.

Using „4-color mode" this can be accomplished by:

```
   WAITVID ( $FF_1F_07_00, %%0123 )
```

This is just ONE saw tooth followed by zero values; but we easily can generate 4 "saw teeth":

```
   WAITVID ( $FF_1F_07_00, %%0123012301230123 )
```

But there is a further configuration parameter („frame clock", in the second video register **VSCL**), allowing to reduce the number of output patterns. Setting "frame clock" to 4 means that only the most right 4 (or 8) bits from the PIXELS are processed before the **WAITVID** instruction releases the video logic again; the "register frame" has now a length of 4 rather than 16.

This all sounds a little bit complicated .. What is the advantage when compared to some simple OUTA instructions?

Very little: It can work faster, and you do not have to take special care for timings and wait times. WAITVID has got its name for a very good reason. Before it starts shifting out the operands it waits for the end of a **previous video operation**! The **next** instruction is fetched as soon as the new shifting has been started. This is handy!

```
' ex10A
VSCLpreset   LONG  $1_004
VCFGpreset   LONG  %0_01_1_00_000_00000000000_0XX_0_11111111

'please do look up the meaning of all those parameters!!

colors       LONG $FF1F0700
   MOV VSCL, VSCLpreset
   MOV VCFG, VCFGpreset
LOOP
   WAITVID colors, #%%0123
   JMP #loop
```

This 4-byte saw tooth is output at pins XX*8 to XX*8+7 at most(!) every 8 clocks = 100 ns/4 bytes = 40 MB/s. This is not bad! Note that the limit is NOT the – still unknown – clocking of the video logic, but our ability to control this logic with instructions!

When we want to output more ambitious signals, we either have to compute them or load them from HUB; this slows things down further!

```
'ex10B
loop
  RDLONG colors, address
  WAITVID colors, #%%0123
  ADD address, #4
  DJNZ period, #loop
```

Though slowed down, this still yields 10 MB/s

But a “hand made” loop is not necessarily MUCH slower:

```
'ex10C
LOOP
  RDBYTE val, address
  SHL    val , #ioPinPos
  MOV    OUTA, val
  ADD    address, #1
  DJNZ   period, #loop
```

This is around 3 MB/s.

When we want later to generate video (4- 6 MHz) or true VGA (25-30 MHz), we already see that we are working at the frontier. Using a “register frame” of just 4 will not do; we shall need a frame length of 8, 16 or even 32 (in “2-color-mode”)

But why do I sound that this could be an issue? Why don't we use a "register frame" of 32 all the time in the first place?

Think! When shifting 32 bits in 2-color-mode the only option is to either output byte A or B from the COLOURS register: "Black" or "White" (or whatever you have stored there). The choice widens when using 4-color mode, but you are restricted to those patterns in the COLORS register for the whole register frame. This is severe constraint as soon as you use a frame size above 4!

Here is another extreme example, we use just **one** I/O pin rather than all 8

```
'ex10E
VSCL_preset   LONG  $1_020
VCFG_preset   LONG  %0_01_0_00_000_00000000000_0XX_0_00000001
colors        LONG  $01_00

   MOV VSCL, VSCL_preset
   MOV VCFG, VCFG_preset
loop
   RDLONG  data32, address
   WAITVID colors, data32
   ADD     address, #4
   DJNZ    length, #loop
```

We can output this bit stream with 32 bit per 400 ns, i.e. 80 Mbit/s per channel (or 40 Mbit/s per two channels in “4-color-mode”).

Someone may still wonder how to use the clock to drive the video logic. As seen above we can use the system clock for 80 MHz. It generally makes no sense to use a much higher clock rate, as we can no longer feed the video logic without disruptions; this would lead to unwanted “bursts” and “jitter”… We have to always adjust the video clock so that there will be a **small** wait left for a new WAITVID!

Sidetrack G: How to program Timer A
Programming timer A is done setting **CTRA** and **FRQA**, e.g.:

```
  MOVI    FRQA, #56           '56*80/512 = about 8,75 MHz
  MOVI    CTRA, #%00001_101   'internal,  PLL = *16/4 = *4 = 35 MHz
```

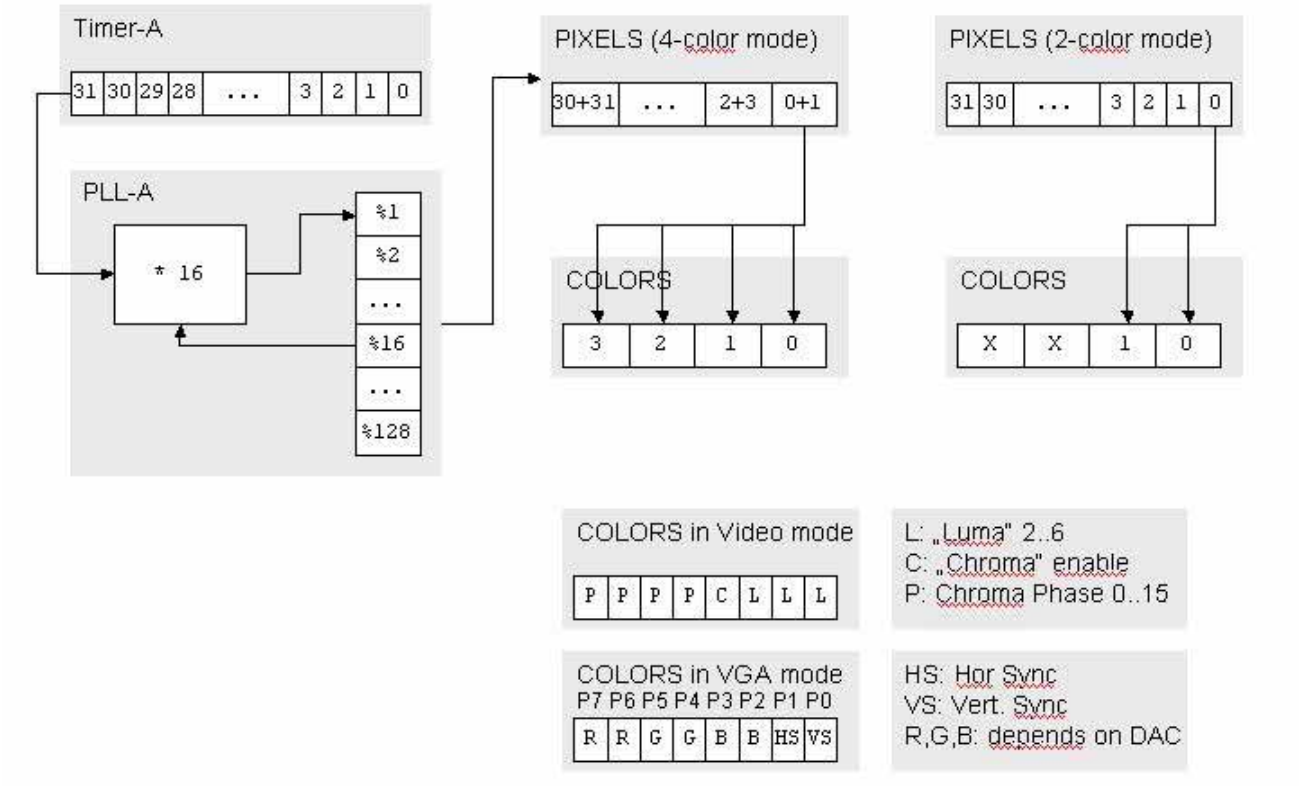
Need an explanation? All right: We use the [b]internal[/b] timer mode, so the timer signal (bit 31 of PHSA) is NOT output to any I/O pin but rather connected to shift bits out of the PIXELS register. This frequency is determined by the value of FRQA, which is added each system clock tick onto PHSA.

Logically this derived clock is always a fraction of the system clock, but this derived clock is used to control a PLL circuit multiplying the clock by 16! In addition to this boost, the PLL will compensate for any jitter if the fraction programmed for the timer is an “odd” number. PLLS work only within certain ranges: the datasheet says: 4 to 8 MHz (which means the clock at PLL output is 64 to 128 MHz).

However we will generally not use THAT clock, but a derived clock by dividing this frequency by 2 (32 – 64 MHz), 4, 8, 16, 32, 64, or 128 (500-1000 kHz).

To program a timer (i.e. the FRQA register) we often use the MOVI instruction, when the value can be “so la la” (1% off). Note that we have to calibrate the crystal and consider temperature anyway when we want to do much better! A ONE in bit 23 results in an Timer overflow after 512 steps = 80 MHz/512 = 156 kHz. This is not good for the PLL (at least according to the datasheet – in practice it will do fine); 25 is the minimum value. The given value in ex10 (=56) will yield 8,75 MHz before we enter the PLL.

The next Diagram 3 will show these relations (timer and video logic) in a simple sketch; please ignore the 4 boxes in the lower right corner; they are needed in the next chapter only.



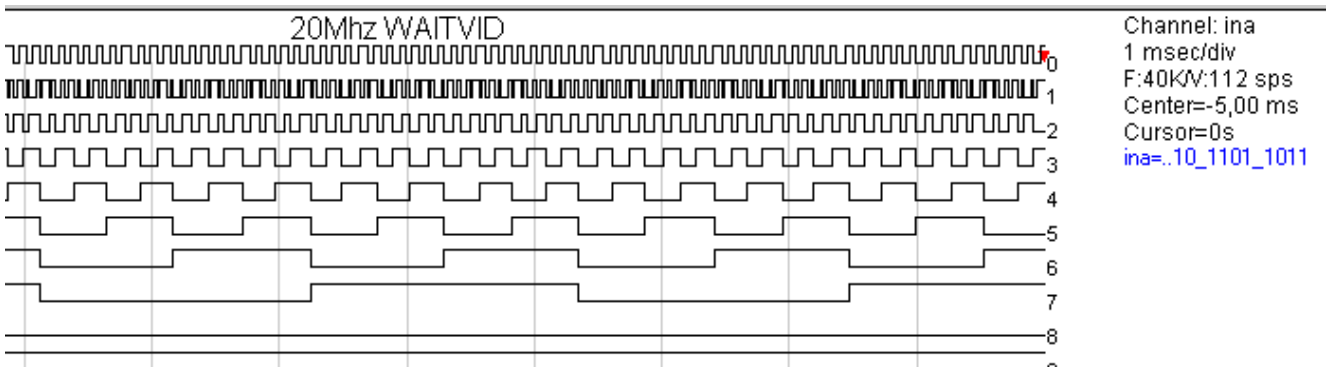
I have to apologize that there had been no complete examples up to now. The reason is, they would have made no sense. If you want to just copy code to see something happen you can as well use one of the many video drivers. But now you should have developed enough understanding of what is going on!

The name of this chapter was: "Video without Video". No kidding: We will NOT have video now! Rather we will "count up" a set of 8 I/O pins (0..7), as slow as possible using the "video logic". And don't be surprised: as we are not interested in speed we can easily use SPIN for the next examples. (You know SPIN, don't you??)

```
'ex11A
CON
  _clkmode = xtal1 + pll8x
  _xinfreq = 10_000_000

  _pinGroup = 0
  _pinMask = $FF<<(_pinGroup*8)

PUB  Main | n
   DIRA := _pinMask
   CTRA := %00001_000 << 23  ' internal, PLL % 128
   FRQA := POSX/CLKFREQ*4_000_000

   VSCL := $1_004            ' register frame = FOUR elements
   VCFG := %0_01_0_00_000_00000000000_000_0_11111111 +
           (_pinGroup << 9)  ' VGA, 2-color-mode
   REPEAT
      WAITVID (n++ ,0)
```

A cheap frequency counter will show us 240 Hz at pin 7 (MSB). My intention was to output just ONE element per frame (VSCL := $1_001), but the signal looked not very stable in that case. I think I have just spotted the reason: Just ONE element per frame has driven WAITVID to 240 Hz *4 *128 = 128 kHz corresponding to 8 us which is below the execution time of the SPIN loop!

As I just got crazy about the new **ViewPort** tool, I have to include a screenshot (Diagram 4):



```
But we can do much slower!
'edx11B
   VSCL := $ff_0ff  'now just 1 element/frame but stretched by 255

The frequency counter at pin 7 (MSB) now shows 4Hz (= 240/255*4)

In ex11A we shifted out 4 (equal) elements per register frame,
which took the four-fold time. We could also shift out all 32
(equal) elements of PIXELS, but the field sizes in VSCL would not
allow this (only 12 bits for "frame clock"); so we have to
restrict ourselves to 16 elements.
'ex11C
   VSCL := $ff_ff0 '16 elements (frame length), 255-fold stretched

As expected the MSB (pin 7) now outputs at a quarter Hz

Last and least we shall challenge the slowest frequency of the
timer the PLL will work with. I still succeeded at 65 kHz (the
data sheet said: 4 MHz!)
'ex11D
   FRQA := POSX/CLKFREQ*65_000

There are few frequency counters showing anything at pin 7 now.
But an LED connected to pin 0 virtuously blinks in a two second
rhythm. (16*255*16/65_000 = 1 s)
```

Interlude 3: What Video is all about: A very gentle approach

Among recent high-tech devices a TV set is most likely the stupidest thing you can imagine. You have to tell it EVERYTHING in most detail! Not once, not twice, no, thirty times – each and every second! European TV sets are somewhat smarter; they only need to be reminded 25 times per second to their task.

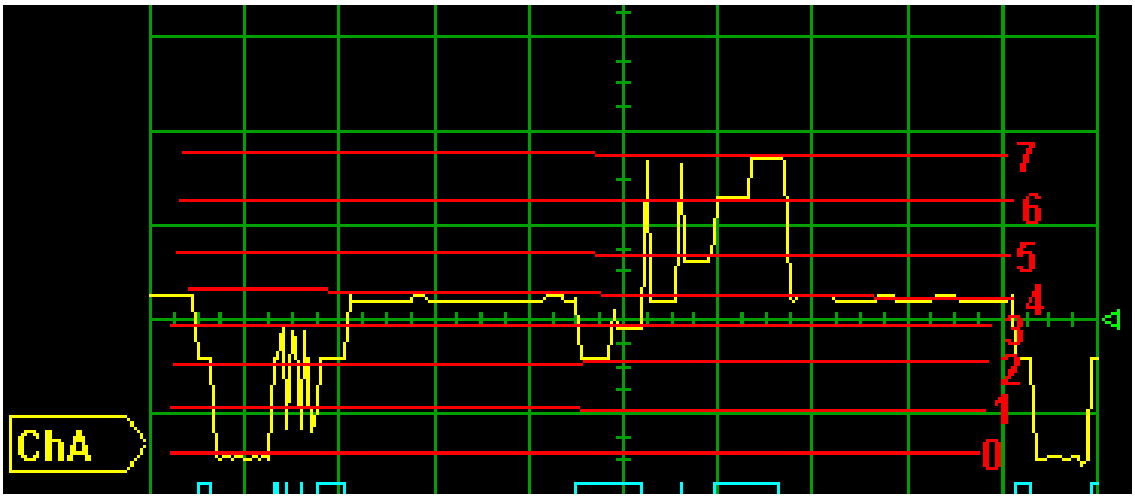
This is no coincidence: TV technology was invented somewhere between Morse's Telegraph and Bell's Telephone. The first TV standards where already established around 1928 (30 lines, 12.5 Hz screen refresh)

I can only think of one device stupider than a TV set: A (non multi-sync) VGA monitor.

But how do they work at all? Well in early times a beam of electrons was focussed on the other - more or less plane and transparent - end of a large radio tube. This end had been coated with some phosphorescent material that transmuted the electronic power into photonic light (Poetic – isn't it?) People were magically attracted by this effect as many are still today.

This beam took its way from the upper left corner to the lower right corner, covering the whole "screen" in exactly the way we read or write in most European languages.

During this path (which according to today's standards takes exactly 16.683 ms) its "luminosity" is changed, so a differentiated image can be displayed. The signal you will need to control the beam looks like the next Diagram 5:



(It is only a small part of the 16 ms – just one “line” of very few pixels and of just a few us. And thanks to Andy, who prepared it!)

Such kind of signal was originally directly applied to one of the grids of the tube.

In contrast to a popular prejudice, electrons rarely move with light speed. So when the beam returns from right to left its “beam power” had to be reduced as much as possible, down to “super black” so to speak; this phase is called “horizontal sync”, followed by a “porch” until the beam has reached its new starting position at the left hand side. There has to be a similar gap in the signal when the beam is moving from bottom right to top left every 1/60 second.

When you want the Prop to produce “video” you have to generate exactly that kind of signal – nothing else at all, no feed back, no input, no test, no check, no if: It will be the most straightforward task in your career as a programmer. Of course you have to know the rules

There was no such thing as a “pixel” at that time. The size of the spots of light emitted from the screen depended on the tubes capability to “focus” the beam. In Y-direction the TV set had to care for the “spacing” of the line, so the number of lines had to be “implemented” in the devices and thus to be standardized early; in X-direction the bandwidth of the video amplifiers set a limit. The levels of the luminosity however stayed “analogue”.

The modern NTSC-M standard defines 720 horizontal and 486 vertical visible pixels.

With the advent of Colour TV “pixels” became more obvious to the public: for each of those formerly somewhat theoretical pixels three coloured dots had to be etched onto the tube. (Needed a lot of busy Chinese, I think!) Of course they couldn’t use less pixels than in the black and white standard for such an expensive device!

And the new Colour TV had to be “compatible” in both directions, that meant: a "monochrome" TV should not be disturbed by “colour signals”, and a Colour TV should be able to live happily without colour.

The solution for this can be considered as one of the magic moments of mankind!

When you look at the 'Diagram 3 you have little hints of colours (I shall try to find some more instructive images, alas my equipment does not allow me to catch those phantastic oscillograms myself at the moment). What you see is the “levels of grey”, the "luma"-signal. The ingenious idea now was to just add a small digital (!) signal – on the Prop adding one "level". However this signal was not locked in the framework of the “luma pixels”, but time shifted (also called: “phase” shifted) a little bit. The amount of this “jitter” or “displacement” was (and is!) used to define a “colour”: No

displacement means “blue”, etc. Obviously the basic clock of this "chroma signal" has to be very precise, the value is: 3.579.545,00 Hz - and that's not kidding! Note the ",00"!

This specific feature made me say TV sets are somewhat more intelligent than VGA monitors are 

To generate this displacement of the "chroma signal" is a little bit tricky to accomplish, and this is the “little magic” going on in the COG’s video logic: You just say: “Blue”, and that’s is!

A black and white TV wouldn’t care, even wouldn’t notice this at all! A colour TV will have to make aware that those displaced signals have to be spotted for. But displaced to what reference? For this reason at the beginning of each line - in the midst of the “back porch” - an “awareness signal” is generated, called “colour burst”. The TV set will sync to it (using a PLL in the old days) to detect the displacements.

This “colour burst” however is NOT part of the “magic” – you have to generate it yourself in your home made driver…

There are more standards all over the world but NTSC-M: PAL and (the very similar) SECAM are most notable. As most small TV sets or car monitors are produced in countries using NTSC (No, I do NOT refer to the United States :-)), they understand this as their “mother language”. PAL is often an “add on”. It is different with the living-room TV. For the Propeller this (and “other formats” as 16:9) is most annoying. Much of the simplicity of video generation gets lost when you try to consider and care for multiple formats.

We also have to understand another more technical detail called “interlacing”. The alert reader has long noticed that deSilva has flopped again: "16 ms? That's not 1/30 second!"
Oh, dear! But you shall find enlightenment soon: The standard requires to transmit the screen contents 30 times per second (29.97 times to be very precise). Doesn’t that flicker? You bet it does! One could choose special screen coating for high afterglow (=”persistence”), as you most likely have seen on RADAR screens. But this is no real solution for fast changing images. A better solution was to split the screen lines (the "frame") into two groups – odd ("upper field") and even ("bottom field") lines – and transmit (and display) them one after another, each 243 double spaced lines taking 1/60 seconds. This uses afterglow “a little bit”, and most people now notice flicker only from the corner of their eye.

This had consequences for the producers of small and cheap monitors. They could successfully develop the attitude that only 243 lines matter and reduced the (expensive) TFT cells to 234 lines (it is unclear why exactly 234, maybe it started with a typo?). As this would result in very “un-square” pixels, they boldly also cut the 720 horizontal pixels in halves; 320 is used in most of those devices today, adapting to 16:9 formats increased this again, to 480 horizontal pixels.

This is what your Propeller displays to in many cases: 320x 234 or 480x 234.

Pictures look best when exactly adapting to this format. Be extremely careful with “interlacing”. It will reduce quality considerably on those monitors. Interlacing can improve quality on large living-room TV sets and some high end car monitors (> $100) sporting true 640x480 pixels. PC monitors will rarely have a video input. Frame catcher cards or USB frame catchers are tricky – you have to consult their manual, but not always with success…

The End

No, this is NOT the end!
DeSilva has many ideas how to continue:
- Best Practices
- Efficient Use of multiple COGs
- Time vs. Space
- Debugging with Ariba's PASD

Also there are still three half-bred sections missing, "The end of video", some remarks wrt the NR post-fix, and a sidetrack "Tricks with OUTB"

But he will have two evening classes about Propeller Programming in the next months, having to be prepared - and the material used there will not be in English…

When time is left, deSilva will re-activate his Multi-Prop project again: A "stackable" low-cost system for "number crunching". Photographs will follow…

Appendix: Pitfalls of SPIN

Real Programmers don't use SPIN!? O yes, they do! SPIN is extremely handy for slow applications. However it has it's drawbacks and pitfalls also (and especially!) a machine programmer must be aware of:

Scope

-1) The rules are relatively simple: NO OVERLOADING! NO SHADOWING!

-2) There is no other possibility to access variables of other objects but using GETters and SETters, however they spend considerable time, in contrast to modern OOP design, where you find a tendency to offer them "for free" (i.e. compilers generate inline code).

-3) GETting an **address** is fine and in the spirit of SPIN as being a "structured assembly language" 
This pointer needs not necessarily address an "array", but can point to any place in VAR or DAT space (but see the next section for further pitfalls)

Memory Allocation

-4) VAR variables are **resorted** by the compiler: LONGS first, follow WORDs, follow BYTES; unawareness of this can lead to deep frustration 

-5) In contrast DAT variables are **padded** if appropriate!

-6) Never forget: VAR is "object space"; only DAT is "global"!

Tree of Objects

-7) Each time you define a name in the OBJ section a new object is "instantiated". That means a new set of VAR memory is (statically) allocated. DAT and CODE always stays the same.

This is extremely frustrating when you have "library objects" used from multiple spots of your program. Take **Float32**. You may need it from the main object and some "sub-object" (maybe **FloatString**). FloatString normally uses the independent and slow FloatMath; so you are inclined to change that to Math32 as well. This is where your problems start ☺ But after you understand their root, you can easily fix Math32 (2 variables in VAR -> DAT)

-8) There is a bug – at least according to my opinion – in COGNEW, as it does not deliver the object context to the SPIN Interpreter in the new COG, which means you can only use procedures from your own object.

```
'Example
  OBJ
      Sub1 : "sub1"
  PUB main
      COGNEW (sub1.go(0),@stack)
'Does not work, and there is no warning
```