PropForth

An Introduction to Interactively programming parallel processors

Sal Sanci copyright 2012, all rights reserved.

1 Document Status

1.1 Documentation Plan

1.2 Revision History

1.2.1 Revision: 2012-03-04_07:00:00

1.2.2 Revision: 2012-04-13_08:00:00

1.2.3 Revision 2012-04-14_12:00:00

1.2.4 Revision 2012-11-15_01:00:00

2 Introduction

2.1 Who is this written for?

2.2 What is PropForth?

2.3 Document Conventions

2.3.1 sc

2.4 Propforth Version

2.5 Loading PropForth

2.6 Teraterm / terminal programs

3 Getting started

3.1 Numbers

3.2 Words

3.3 Stack

3.4 Forth Words in detail- Part 1

3.4.1 sc

3.4.2 cr
3.4.3 . (DOT)
3.4.4 u
3.4.5 +
3.4.6 -
3.4.7 *
3.4.8 /
3.5 Defining new words
3.6 Memory
3.6.1 Cog Memory
3.6.2 Main Memory
3.7 Forth Words in detail - Part 2
3.7.1 hex
3.7.2 decimal
3.7.3 COG@
3.7.4 COG!
3.7.5 L@
3.7.6 L!
3.7.7 W@
3.7.8 W!
3.7.9 C@
3.7.1 C!
3.7.2 and
3.7.3 andn
3.7.4 invert
3.7.5 lshift
3.7.6 rshift
3.7.7 or
3.7.8 xor
3.7.9 >
3.7.10 =
3.7.11 <
3.7.12 <>
3.8 Strings
3.9 Forth Words in detail - Part 3
3.9.1 .cstr

3.9.2 ."
3.9.3 String examples
3.10 Control Flow
3.10.1 Introduction to if --- then
3.10.2 Introduction to if --- else --- then
3.10.3 Introduction to begin --- until
3.10.4 Introduction to do --- loop
3.11 Forth Words in detail - Part 4
3.11.1 if --- then
3.11.2 if --- else --- then
3.11.3 begin --- until
3.11.4 do --- loop
3.12 Talking to the other cogs
3.13 Forth Words in detail - Part 5
3.13.1 >con
3.13.2 cogx
3.13.3 cogreset
3.13.4 Underflow error
3.13.5 .const?

4 PropForth pin I/O
4.1 Forth Words - Part 5
4.1.1 space
4.1.2 delms
4.1.3 pinin
4.1.4 pinout
4.1.5 pinlo
4.1.6 pinhi
4.1.7 px
4.1.8 px?
4.1.9 pins?
4.1.10 I/O example 1
4.1.11 I/O example 2

5 LogicAnalyzer
5.1 Loading lac

5.2 Running lac

6 PropForth Input/Output

6.1 Overview

6.2 Technical details

6.2.1 Input Word

6.2.2 Output Pointer

6.2.3 Advanced topics

7 PropForth Assembler

7.1 Kernels and Assembler Words

7.2 Writing Assembler Words

7.3 The Stack in Detail

7.4 Assembler Mnemonic List

7.5 Assembler Prefix List

7.6 Assembler Postfix List

8 A Low Power example

9 Forth Word Reference

9.1 Core Word Set (build_BootKernel, build_BootOpt)

9.1.1 #

9.1.2 #>

9.1.3 #s

9.1.4 $C_IP

9.1.5 $C_a_(+loop)

9.1.6 $C_a_(loop)

9.1.7 $C_a_0branch

9.1.8 $C_a_2>r

9.1.9 $C_a_branch

9.1.10 $C_a_doconw

9.1.11 $C_a_dovarw

9.1.12 $C_a_exit

9.1.13 $C_a_litl

9.1.14 $C_a_litw

9.1.15 $C_a_lxasm
9.1.16 $C_a_next
9.1.17 $C_fMask
9.1.18 $C_resetDreg
9.1.19 $C_varEnd
9.1.20 $H_cogdata
9.1.21 $H_cq
9.1.22 $H_dq
9.1.23 $H_entry
9.1.24 $S_baud
9.1.25 $S_cdsz
9.1.26 $S_con
9.1.27 $S_rxpin
9.1.28 $S_txpin
9.1.29 '
9.1.30 (+loop)
9.1.31 (createbegin)
9.1.32 (createend)
9.1.33 (fl)
9.1.34 (flout)
9.1.35 (ioconn)
9.1.36 (iodis)
9.1.37 (iolink)
9.1.38 (iounlink)
9.1.39 (loop)
9.1.40 (nfcog)
9.1.41 (prop)
9.1.42 (version)
9.1.43 +
9.1.44 +loop
9.1.45 -
9.1.46 -1
9.1.47 .
9.1.48 ."
9.1.49 ...
9.1.50 .cstr
9.1.51 .str

9.1.52 .strname

9.1.53 0

9.1.54 0<

9.1.55 0<>

9.1.56 0=

9.1.57 0>

9.1.58 0>=

9.1.59 0branch

9.1.60 1

9.1.61 1+

9.1.62 1-

9.1.63 2

9.1.64 2+

9.1.65 2-

9.1.66 2/

9.1.67 2>r

9.1.68 2drop

9.1.69 2dup

9.1.70 2lock

9.1.71 2unlock

9.1.72 3drop

9.1.73 4*

9.1.74 4+

9.1.75 :

9.1.76 ;

9.1.77 <

9.1.78 <#

9.1.79 <=

9.1.80 <>

9.1.81 =

9.1.82 >

9.1.83 >=

9.1.84 >con

9.1.85 >in

9.1.86 >m

9.1.87 >out

9.1.88 >r

9.1.89 C!
9.1.90 C@
9.1.91 C@++
9.1.92 COG!
9.1.93 COG@
9.1.94 ERR
9.1.95 L!
9.1.96 L@
9.1.97 RS!
9.1.98 RS@
9.1.99 ST!
9.1.100 ST@
9.1.101 W!
9.1.102 W+!
9.1.103 W@
9.1.104 [if
9.1.105 [ifdef
9.1.106 [ifndef
9.1.107 \
9.1.108]
9.1.109 _accept
9.1.110 _asmpfa>nfa
9.1.111 _cnip
9.1.112 _dictsearch
9.1.113 _dl
9.1.114 _ecs
9.1.115 _eeread
9.1.116 _eewrite
9.1.117 _femit?
9.1.118 _finit
9.1.119 _fkey?
9.1.120 _forthpfa>nfa
9.1.121 _if
9.1.122 _lc
9.1.123 _lockarray
9.1.124 _maskin
9.1.125 _maskouthi

9.1.126 _maskoutlo
9.1.127 _mmcs
9.1.128 _p+
9.1.129 _p?
9.1.130 _qp
9.1.131 _serial
9.1.132 _sp
9.1.133 _udf
9.1.134 _wc1
9.1.135 _wkeyto
9.1.136 _xasm1>1
9.1.137 _xasm2>0
9.1.138 _xasm2>1
9.1.139 _xasm2>1IMM
9.1.140 _xasm2>flag
9.1.141 _xasm2>flagIMM
9.1.142 _xis
9.1.143 _xnu
9.1.144 accept
9.1.145 alignl
9.1.146 alignw
9.1.147 allot
9.1.148 and
9.1.149 andn
9.1.150 andnC!
9.1.151 asmlabel
9.1.152 base
9.1.153 begin
9.1.154 between
9.1.155 bl
9.1.156 bounds
9.1.157 branch
9.1.158 build_BootKernel
9.1.159 build_BootOpt
9.1.160 c"
9.1.161 c,
9.1.162 cappend

9.1.163 cappendn
9.1.164 ccopy
9.1.165 ccreate
9.1.166 cds
9.1.167 checkdict
9.1.168 clearkeys
9.1.169 clkfreq
9.1.170 cmove
9.1.171 cnt
9.1.172 cogcds
9.1.173 coghere
9.1.174 cogid
9.1.175 cogio
9.1.176 cogiochan
9.1.177 cognchan
9.1.178 cognumpad
9.1.179 cogpad
9.1.180 cogreset
9.1.181 cogstate
9.1.182 cogstop
9.1.183 cogx
9.1.184 compile?
9.1.185 cq
9.1.186 cr
9.1.187 create
9.1.188 cstr=
9.1.189 delms
9.1.190 dictend
9.1.191 dira
9.1.192 do
9.1.193 doconl
9.1.194 doconw
9.1.195 doloop
9.1.196 dothen
9.1.197 dovarl
9.1.198 dovarw
9.1.199 dq

9.1.200 drop
9.1.201 dup
9.1.202 else
9.1.203 emit
9.1.204 exec
9.1.205 execute
9.1.206 execword
9.1.207 exit
9.1.208 femit?
9.1.209 fill
9.1.210 find
9.1.211 fisnumber
9.1.212 fkey?
9.1.213 fl
9.1.214 fl_in
9.1.215 fl_lock
9.1.216 fnumber
9.1.217 forthentry
9.1.218 freedict
9.1.219 fstart
9.1.220 here
9.1.221 herelal
9.1.222 herewal
9.1.223 hex
9.1.224 hubopf
9.1.225 hubopr
9.1.226 i
9.1.227 if
9.1.228 immediate
9.1.229 ina
9.1.230 init_coghere
9.1.231 initcon
9.1.232 interpret
9.1.233 interpretpad
9.1.234 io
9.1.235 ioconn
9.1.236 iodis

9.1.237 iolink
9.1.238 iounlink
9.1.239 isdigit
9.1.240 isnamechar
9.1.241 isnumber
9.1.242 isunumber
9.1.243 key
9.1.244 l,
9.1.245 l>w
9.1.246 lasterr
9.1.247 lastnfa
9.1.248 leave
9.1.249 litl
9.1.250 litw
9.1.251 lock
9.1.252 lockdict
9.1.253 loop
9.1.254 lshift
9.1.255 lxasm
9.1.256 max
9.1.257 memend
9.1.258 min
9.1.259 name=
9.1.260 namecopy
9.1.261 namelen
9.1.262 namemax
9.1.263 negate
9.1.264 nextword
9.1.265 nfa>lfa
9.1.266 nfa>next
9.1.267 nfa>pfa
9.1.268 nfcog
9.1.269 nip
9.1.270 npfx
9.1.271 number
9.1.272 numpad
9.1.273 numpadsize

9.1.274 onboot

9.1.275 onreset

9.1.276 or

9.1.277 orC!

9.1.278 orInfa

9.1.279 outa

9.1.280 over

9.1.281 pad

9.1.282 pad>in

9.1.283 pad>out

9.1.284 padbl

9.1.285 padsize

9.1.286 par

9.1.287 parse

9.1.288 parsebl

9.1.289 parsenw

9.1.290 parseword

9.1.291 pfa>nfa

9.1.292 prop

9.1.293 propid

9.1.294 r>

9.1.295 reboot

9.1.296 reset

9.1.297 rot

9.1.298 rot2

9.1.299 rshift

9.1.300 serial

9.1.301 seti

9.1.302 skipbl

9.1.303 space

9.1.304 spaces

9.1.305 state

9.1.306 swap

9.1.307 t0

9.1.308 t1

9.1.309 tbuf

9.1.310 then

9.1.311 thens
9.1.312 tochar
9.1.313 todigit
9.1.314 tuck
9.1.315 u*
9.1.316 u
9.1.317 u/
9.1.318 u/mod
9.1.319 u>=
9.1.320 um*
9.1.321 um/mod
9.1.322 unlock
9.1.323 unlockall
9.1.324 until
9.1.325 unumber
9.1.326 version
9.1.327 w,
9.1.328 w>l
9.1.329 wconstant
9.1.330 wlastnfa
9.1.331 wvariable
9.1.332 xisnumber
9.1.333 xnumber
9.1.334 xor
9.1.335 {
9.1.336 }

9.2 DevKernel Word Set (BuildDevKernel)
9.2.1 #C
9.2.2 $C_a__xasm2>1
9.2.3 $C_a__xasm2>1IMM
9.2.4 $C_a_doconl
9.2.5 $C_a_dovarl
9.2.6 $C_rsPtr
9.2.7 $C_rsTop
9.2.8 $C_stPtr
9.2.9 $C_stTOS

9.2.10 $C_stTop
9.2.11 (dumpb)
9.2.12 (dumpe)
9.2.13 (dumpm)
9.2.14 (forget)
9.2.15 *
9.2.16 */
9.2.17 */mod
9.2.18 .byte
9.2.19 .cogch
9.2.20 .con
9.2.21 .conbyte
9.2.22 .concr
9.2.23 .concstr
9.2.24 .conemit
9.2.25 .conlong
9.2.26 .const?
9.2.27 .conwait
9.2.28 .conword
9.2.29 .long
9.2.30 .word
9.2.31 /
9.2.32 /mod
9.2.33 1lock
9.2.34 1unlock
9.2.35 2*
9.2.36 4-
9.2.37 4/
9.2.38 EC@
9.2.39 EW!
9.2.40 EW@
9.2.41 _bf
9.2.42 _eestart
9.2.43 _eestop
9.2.44 _ft
9.2.45 _lf
9.2.46 _nd

9.2.47 _pna
9.2.48 _sclh
9.2.49 _scli
9.2.50 _scll
9.2.51 _sclo
9.2.52 _sda?
9.2.53 _sdah
9.2.54 _sdai
9.2.55 _sdal
9.2.56 _sdao
9.2.57 _wf
9.2.58 _words
9.2.59 abs
9.2.60 andC!
9.2.61 build?
9.2.62 build_DevKernel
9.2.63 cog?
9.2.64 cogdump
9.2.65 constant
9.2.66 decimal
9.2.67 dump
9.2.68 edump
9.2.69 eereadpage
9.2.70 eewritepage
9.2.71 forget
9.2.72 free
9.2.73 ibound
9.2.74 invert
9.2.75 io>cogchan
9.2.76 j
9.2.77 lasti?
9.2.78 lock?
9.2.79 onreset
9.2.80 pfa?
9.2.81 pinhi
9.2.82 pinin
9.2.83 pinlo

9.2.84 pinout
9.2.85 px
9.2.86 px?
9.2.87 rev
9.2.88 revb
9.2.89 rnd
9.2.90 rndtf
9.2.91 rs?
9.2.92 saveforth
9.2.93 sc
9.2.94 serflags?
9.2.95 sersendbreak
9.2.96 sersetflags
9.2.97 sign
9.2.98 st?
9.2.99 u*/
9.2.100 u*/mod
9.2.101 variable
9.2.102 waitcnt
9.2.103 waitpeq
9.2.104 waitpne
9.2.105 words

9.3 EEpromKernel Word Set (build_fsrd, build_fswr)
9.3.1 _fnf
9.3.2 _fsfind
9.3.3 _fsfree
9.3.4 _fsk
9.3.5 _fslast
9.3.6 _fsload
9.3.7 _fsnext
9.3.8 _fsp
9.3.9 _fspa
9.3.10 _fsrd
9.3.11 _fsread
9.3.12 _fswr
9.3.13 build_fsrd

9.3.14 build_fswr
9.3.15 fsbot
9.3.16 fsclear
9.3.17 fsdrop
9.3.18 fsfree
9.3.19 fsload
9.3.20 fsls
9.3.21 fsps
9.3.22 fsread
9.3.23 fstop
9.3.24 fswrite
9.3.25 onboot

9.4 SDKernel Word Set (build_sd)
9.4.1 $S_sd_clk
9.4.2 $S_sd_cs
9.4.3 $S_sd_di
9.4.4 $S_sd_do
9.4.5 .num
9.4.6 _fnf
9.4.7 _fsk
9.4.8 _nf
9.4.9 _readlong
9.4.10 _sd_alloc
9.4.11 _sd_appendbytes
9.4.12 _sd_ccs
9.4.13 _sd_clk_out
9.4.14 _sd_clk_out_h
9.4.15 _sd_clk_out_l
9.4.16 _sd_cmdr16
9.4.17 _sd_cmdr40
9.4.18 _sd_cmdr8
9.4.19 _sd_cmdr8data
9.4.20 _sd_cogend
9.4.21 _sd_cs_out
9.4.22 _sd_cs_out_h
9.4.23 _sd_cs_out_l

9.4.24 _sd_di_out
9.4.25 _sd_di_out_h
9.4.26 _sd_di_out_l
9.4.27 _sd_dn
9.4.28 _sd_do_in
9.4.29 _sd_fsp
9.4.30 _sd_hash
9.4.31 _sd_hc
9.4.32 _sd_init
9.4.33 _sd_initdir
9.4.34 _sd_initialized
9.4.35 _sd_maxblock
9.4.36 _sd_readdata
9.4.37 _sd_setdirentry
9.4.38 _sd_shift_in
9.4.39 _sd_shift_inlong
9.4.40 _sd_shift_out
9.4.41 _sd_shift_outlong
9.4.42 _sd_writedata
9.4.43 a_shift
9.4.44 build_sd
9.4.45 cd
9.4.46 cd.
9.4.47 cd/
9.4.48 cog>mem
9.4.49 cog>pad
9.4.50 cog>tbuf7
9.4.51 cwd
9.4.52 fcreate
9.4.53 fload
9.4.54 fread
9.4.55 fstat
9.4.56 fwrite
9.4.57 ls
9.4.58 mem>cog
9.4.59 mkdir
9.4.60 onboot

9.4.61 pad>cog
9.4.62 sd_append
9.4.63 sd_appendblk
9.4.64 sd_blockread
9.4.65 sd_blockwrite
9.4.66 sd_cd
9.4.67 sd_cd.
9.4.68 sd_cogbuf
9.4.69 sd_cogbufclr
9.4.70 sd_createdir
9.4.71 sd_createfile
9.4.72 sd_cwd
9.4.73 sd_find
9.4.74 sd_init
9.4.75 sd_load
9.4.76 sd_loadblk
9.4.77 sd_lock
9.4.78 sd_ls
9.4.79 sd_mount
9.4.80 sd_read
9.4.81 sd_readblk
9.4.82 sd_stat
9.4.83 sd_trunc
9.4.84 sd_uninit
9.4.85 sd_unlock
9.4.86 sd_write
9.4.87 tbuf>cog7
9.4.88 v_currentdir
9.4.89 v_sd_clk
9.4.90 v_sd_di
9.4.91 v_sd_do
9.4.92 v_sdbase

1 Document Status

1.1 Documentation Plan

The long term plan is to have this document be maintained by the user community. It is structured so that we can add sections to deal with specific projects / PropForth capabilities. It will require an editor, but as time progresses this should be a lighter task.

This Document is currently produced using Word 2007. It uses specific styles for formatting to allow publishing to a web format, as well as other electronic formats.

1.2 Revision History

1.2.1 Revision: 2012-03-04_07:00:00

This revision is draft. It needs to be proof read and edited.

1.2.2 Revision: 2012-04-13_08:00:00

Changed Word options so smart quotes are not used, and - is not changed to dash. This should ensure you can copy from the html version to a text document or terminal screen with proper fidelity. Re: issue 145

1.2.3 Revision 2012-04-14_12:00:00

Fixed some remaining - (dash) problems.

1.2.4 Revision 2012-11-15_01:00:00

Removed useless page numbers in contents. Enhanced visual aspects of readability. Minor clarification of some sub-headings.

---

2 Introduction

2.1 Who is this written for?

PropForth is written for people who want to interactively use and program Propeller based systems. It assumes you are familiar with Propeller based hardware, and have some knowledge of the Propeller architecture.

2.2 What is PropForth?

PropForth runs on the propeller. It is an interactive development and run time system. You can communicate with PropForth using a standard serial port, or via ethernet and telnet depending on the capabilities of the system you are using. It has a rich set of words which allow you to

quickly and easily manipulate the hardware interactively. It is easy to program new words and use them in addition to the native set of words. Where necessary, you can write new words in assembler code for fast execution.

PropForth is outstanding for rapid exploration and development. The interactive nature allows you to try new things easily, verify functions, and allows for easy monitoring of hardware. You can manipulate IO pins on one cog, and user another cog to monitor the pins.

Each Propeller can run PropForth on up to 8 cogs, and you can concurrently interact with the cogs. Normal configuration uses one cog for serial or ethernet communication allowing for up to 7 concurrent PropForth sessions.

It is based on the Forth programming language, but it has been tuned for the propeller architecture and parallel execution. PropForth does not prevent or protect you from manipulating the propeller however you like. While it can be simple to use, it is also very powerful.

2.3 Document Conventions

This document contains a lot of code samples and the following are the conventions used.

For Comments:

\ A code comment, frequently documenting the stack use of the word.

The backslash denotes a comment. When a backslash is encountered, the interpreter ignores the rest of the characters on the line (the CR at the end of the line ends the comment). Comment lines are green in this document. A comment is entered for each command word definitions, and includes values needed on the stack (inputs), and the values to be left on the stack (outputs).

For text typed or pasted in at the prompt:

1 3 -100

Text to be typed or pasted into the command line at the command prompt is blue in this document.

For the output of the PropForth interpreter, and user written words:

Prop0 Cog6 ok

Forth words are documented in the following way. The text in the example input can be copied and pasted to a PropForth terminal

2.3.1 sc

\ sc (--)

Example input

Output line1

More output

2.4 Propforth Version

This text is written assuming PropForth V5.0, the download is available on Google Code. We will be using the PropForth System except where explicitly noted.

2.5 Loading PropForth

Download PropForth, and unzip. In the directory CurrentRelease\PropForth, follow the instructions in GettingStarted.txt.

2.6 Teraterm / terminal programs

The examples were written and tested using Teraterm, however just about any terminal program should work.

3 Getting started

The best way to use this document is with a running PropForth system. Use the examples to explore PropForth. At the end of the getting started section, you will have been introduced to a number of the native PropForth words and you will be able to program new words.

The sections which follow will then focus on different topics and continue the exploration of PropForth.

3.1 Numbers

At the prompt, numbers can be entered in a few ways.

This is an example of both positive and negative decimal numbers being entered. The _ (underscore) is available for formatting. It is ignored when evaluating a number.

1 3 -100

Prop0 Cog6 ok

Entering a hexadecimal number, often useful, is done by preceding the number with an h. Once again the _ (underscore) is ignored. Note that only caps are used in hexadecimal numbers.

h100 h-100 hFFFF_FFFF

Prop0 Cog6 ok

Binary numbers are preceded with a b.

b_1001_0011 b0010

Prop0 Cog6 ok

Decimal numbers are default at boot, but they can be entered explicitly by preceding the number with a d.

d_1 d3 d-100

Prop0 Cog6 ok

Remember, Hex digits are always UPPERCASE letters A, B, C, D, E, and F; and the number base designators b, d, and h (for binary, decimal, and hex) are always lowercase.

Most variables in PropForth are 16bit variables. This save a lot of space in PropForth, but a 16 bit variable is a positive number between 0 and 65,535. For many uses this is adequate. If you need positive numbers larger than 65,535 or you need negative numbers, you must use longs. PropForth words treat numbers on the stack as signed long numbers by default.

3.2 Words

The PropForth interpreter can understand numbers and words. Words and numbers are delimited by blanks. Word names are up to 31 characters, and can contain any sequence of characters, symbols, and letters. PropForth is extended by programming new words.

The core set of PropForth words includes words like + - * / .

When a new word is defined it can use other words. So a word is really a sequence of other words.

At the low level, some words are a sequence of assembler instruction codes. These are generally native PropForth words. We will see later how we can program words, and later still how we can program assembler words

3.3 Stack

When you enter a number at the prompt, PropForth puts the numbers on a stack. The stack can hold up to 32 32-bit numbers, but the PropForth interpreter code uses the stack as well. The pragmatic limit of numbers on the stack is about 24.

Each cog has its own stack in cog memory.

To see what is on the stack we can use the st? word.

1 3 h-100 st?

ST: 0_000_000_001 0_000_000_003 -0_000_000_256

Prop0 Cog6 ok

By convention the stack is represented with the bottom of the stack on the left and the top of the stack on the right.

Normal documentation of a PropForth word is a comment line which shows how it changes the stack.

\ + (n1 n2 -- n1+n2) \ sum of n1 & n2

On the left of the -- , is the stack before word executes, and on the right, the stack after the word executes. Once again, the rightmost numbers are the top of the stack.

3.4 Forth Words - Part 1

The best way to understand a word is to run the examples. The words are introduced in order of use in the examples. The first comment line is the documentation on the word. The next line is what is typed at the prompt. The remaining is the output.

The examples are ordered to give an introduction to PropForth, and introduce the words and concepts so you can start programming. Feel free to start experimenting whenever you like, it is the purpose of PropForth to help you experiment.

3.4.1 sc

\ sc (--) clears the stack

1 2 3 st? sc st?

ST: 0_000_000_001 0_000_000_002 0_000_000_003

3 items cleared

ST:

Prop0 Cog6 ok

3.4.2 cr

\ cr (--) emits a carriage return

1 2 3 cr cr st? cr cr sc cr cr st?

ST: 0_000_000_001 0_000_000_002 0_000_000_003

3 items cleared

ST:

Prop0 Cog6 ok

3.4.3 . (DOT)

\ . (n1 --) prints the signed number on the top of the stack

sc 1 -2 3 cr st? cr . cr st? cr . cr st? cr . cr st?

0 items cleared

ST: 0_000_000_001 -0_000_000_002 0_000_000_003

3

ST: 0_000_000_001 -0_000_000_002

-2

ST: 0_000_000_001

1

ST:

Prop0 Cog6 ok

3.4.4 u.

\ u. (n1 --) prints the unsigned number on the top of the stack

sc 1 -2 3 cr st? cr u. cr st? cr u. cr st? cr u. cr st?

0 items cleared

ST: 0_000_000_001 -0_000_000_002 0_000_000_003

3

ST: 0_000_000_001 -0_000_000_002

4294967294

ST: 0_000_000_001

1

ST:

Prop0 Cog6 ok

Note in the default mode, which is decimal numbers, st? prints negative numbers as -nnnn. The range of 32bit signed longs is -2147483648 to 2147483647. The range of unsigned numbers is 0 - 4294967295.

3.4.5 +

\ + (n1 n2 -- n1+n2) \ sum of n1 & n2

sc 1 2 3 4 5 -6 cr st? cr + cr st? cr + cr st? cr + cr st? cr + cr st? cr + cr st?

0 items cleared

ST: 0_000_000_001 0_000_000_002 0_000_000_003 0_000_000_004 0_000_000_005 -0_000_000_006

ST: 0_000_000_001 0_000_000_002 0_000_000_003 0_000_000_004 -0_000_000_001

ST: 0_000_000_001 0_000_000_002 0_000_000_003 0_000_000_003

ST: 0_000_000_001 0_000_000_002 0_000_000_006

ST: 0_000_000_001 0_000_000_008

ST: 0_000_000_009

Prop0 Cog6 ok

3.4.6 -

\ - (n1 n2 -- n1-n2) \ subtracts n2 from n1

sc 1 2 3 4 5 -6 cr st? cr - cr st? cr - cr st? cr - cr st? cr - cr st? cr - cr st?

3.4.7 *

\ * (n1 n2 -- n1*n2) n1 multiplied by n2

sc 1 2 3 4 5 -6 cr st? cr * cr st? cr * cr st? cr * cr st? cr * cr st? cr * cr st?

1 items cleared

ST: 0_000_000_001 0_000_000_002 0_000_000_003 0_000_000_004 0_000_000_005 -0_000_000_006

ST: 0_000_000_001 0_000_000_002 0_000_000_003 0_000_000_004 -0_000_000_030

ST: 0_000_000_001 0_000_000_002 0_000_000_003 -0_000_000_120

ST: 0_000_000_001 0_000_000_002 -0_000_000_360

ST: 0_000_000_001 -0_000_000_720

ST: -0_000_000_720

Prop0 Cog6 ok

* is a signed multiply. There is a u* for unsigned multiply.

3.4.8 /

\ / (n1 n2 -- n1/n2) n1 divided by n2

sc 10_000_000 3 cr st? cr / cr st? 10 cr st? cr / cr st? -45 cr st? cr / cr st?

1 items cleared

ST: 0_010_000_000 0_000_000_003

ST: 0_003_333_333

ST: 0_003_333_333 0_000_000_010

ST: 0_000_333_333

ST: 0_000_333_333 -0_000_000_045

ST: -0_000_007_407

Prop0 Cog6 ok

/ is a signed integer divide. There is a u/ for unsigned integer divide.

3.5 Defining new words

The following examples show how to define new words. Once a word is defined, it is

immediately available. This capability allows you try new things quickly . As a rule, defined words should be no more than 20 or 30 lines, most should only be a few lines.

```
\ new word example1

: add37 37 + ; sc 1 2 3 cr st? add37 cr st?
```

0 items cleared

ST: 0_000_000_001 0_000_000_002 0_000_000_003

ST: 0_000_000_001 0_000_000_002 0_000_000_040

Prop0 Cog6 ok

The : (COLON) indicates that we are starting a definition, it is followed by a space, then the name of our new word. We then type what we want the word to do, and end the definition of our new word with a ; (SEMICOLON).

```
\ new word example2

: times9 9 * ; sc 1 2 40 cr st? times9 cr st?
```

3 items cleared

ST: 0_000_000_001 0_000_000_002 0_000_000_040

ST: 0_000_000_001 0_000_000_002 0_000_000_360

Prop0 Cog6 ok

We can use the words we defined just as we use native forth words.

```
\ new word example3

: add37times9 add37 times9 ; sc 1 2 3 cr st? add37times9 cr st?
```

3 items cleared

ST: 0_000_000_001 0_000_000_002 0_000_000_003

ST: 0_000_000_001 0_000_000_002 0_000_000_360

Prop0 Cog6 ok

3.6 Memory

Before we proceed, a short discussion on memory. The propeller has 32k bytes of main memory that is accessible to all the cogs. Each cog has 512 32bit longs which are only accessible to the individual cogs.

3.6.1 Cog Memory

In cog memory, PropForth has the assembler instructions for the interpreter, the space for the stack, and the space for the return stack, temporary variables, and the assembler word page area, which will be discussed later. This takes up 320 longs. There are 16 special registers in each cog, so this leaves 176 long which are available. Cog memory is accessible as 32bit longs.

3.6.2 Main Memory

In main memory, there are 8 special areas of 224 bytes of length each, which are dedicated to the PropForth interpreter running on each cog. The rest of main memory is used by the forth dictionary. The forth dictionary is the forth words and their definitions. When the interpreter gets input from the prompt, it looks up the words in the dictionary, and executes the definition. If it does not find the word definition, it attempts to interpret it as a number. If this succeeds the number goes on the stack.

Variables and constants are also defined in the forth dictionary.

Main memory is accessible as 8bit bytes, 16bit words, or 32bit longs.

3.6.2.1 Variables

Variables in main memory are normally defined as either 16bit words, or 32bit longs.

To define a 16bit word variable:

wvariable name_of_variable

To define a long variable

variable name_of_variable

Most variables in PropForth are 16bit variables. This save a lot of space in PropForth, but a 16 bit variable is a positive number between 0 and 65,535. For many uses this is adequate. If you need positive numbers larger than 65,535 or you need negative numbers, you must use longs.

3.6.2.2 Constants

Like variables, constants in main memory are normally defined as either 16bit words, or 32bit longs.

To define a 16-bit word constant, the value is on top of the stack:

value wconstant name_of_constant

To define a long constant, the value is on the top of the stack:

value constant name_of_constant

3.7 Forth Words - Part 2

3.7.1 hex

\ hex (--) set the base for hexadecimal

sc 0 1 -1 10 100 cr st? cr hex cr st? decimal

0 items cleared

ST: 0_000_000_000 0_000_000_001 -0_000_000_001 0_000_000_010 0_000_000_100

ST: 0000_0000 0000_0001 FFFF_FFFF 0000_000A 0000_0064

Prop0 Cog6 ok

Hexadecimal mode is useful when interfacing to propeller registers or peripheral registers. PropForth can support any base from 2 (binary) to 64. The most common are 10 (decimal) 16 (hex). These have native words that set the base. For other bases, such as 8 (octal) for PDP11 programmers, or 2 (binary), use: n base W! This sets the base to n, and all numbers will be printed in that base. This can give some pretty strange looking numbers but it is very useful as we will see later.

When using n base W! to set the base, remember one cannot use 10 base W! to set the base back to decimal, because 10 represents a different value in each base (10 always represents the base value in every base). Use d_10 base W! or the word decimal.

3.7.2 decimal

\ decimal (--) set the base for decimal

sc hex 0 1 FFFF_FFFF A 64 cr st? decimal cr st?

5 items cleared

ST: 0000_0000 0000_0001 FFFF_FFFF 0000_000A 0000_0064

ST: 0_000_000_000 0_000_000_001 -0_000_000_001 0_000_000_010 0_000_000_100

Prop0 Cog6 ok

3.7.3 COG@

\ COG@ (addr -- n1) \ fetch 32 bit value at cog addr

sc cnt cr st? cr COG@ st? cr u. cnt COG@ cr u. cr

sc cnt cr st? cr COG@ st? cr u. cnt COG@ cr u. cr

0 items cleared

ST: 0_000_000_497

ST: -2_113_416_610

2181550686

2182562366

Prop0 Cog6 ok

The cnt register, in cog memory, is a high speed 32bit counter incrementing once every clock cycle. Normally with a propeller running at 80Mhz, it increments once every 12.5 nanoseconds. This register wraps around to zero about every 53.7 seconds at 80 Mhz.

3.7.4 COG!

\ COG! (n1 addr --) \ store 32 bit value (n1) at cog addr

sc outa cr st? cr COG@ st? cr 1 or outa st? cr COG! st? cr outa COG@ st?

1 items cleared

ST: 0_000_000_500

ST: 0_000_000_000

ST: 0_000_000_001 0_000_000_500

ST:

ST: 0_000_000_001

Prop0 Cog6 ok

Each cog has an outa register, in this case we are reading the value, setting bit 0 to 1, and writing the value. If the dira register bit 0 is set to 1 as well, the value of the outa register, will be or'd with all the other cogs outa registers bit 0 with dira registers bit 0, and appear on io pin 0.

Normally, one cog has a pin set to output, so the value on the io pin is corresponds to the bit in that cog's outa register.

3.7.5 L@

\ L@ (addr -- n1) \ fetch 32 bit value at main memory addr

sc variable tmp 0 tmp cr st? cr L! st? cr tmp st? cr L@ st?

2 items cleared

ST: 0_000_000_000 0_000_017_604

ST:

ST: 0_000_017_604

ST: 0_000_000_000

Prop0 Cog6 ok

3.7.6 L!

\ L! (n1 addr --) \ store 32 bit value (n1) at main memory addr

sc variable tmp1 -1 tmp1 cr st? cr L! st? cr tmp1 st? cr L@ st?

1 items cleared

ST: -0_000_000_001 0_000_017_620

ST:

ST: 0_000_017_620

ST: -0_000_000_001

Prop0 Cog6 ok

32bit long addresses must be long aligned. In this example, the forth word "variable" uses the next text string it encourters (in this case "tmp1") and defines a 32bit variable using that string as the name. Now when tmp1 is entered it returns the address of the variable. This is how we store values to a memory location using a variable name.

3.7.7 W@

\ W@ (addr -- h1) \ fetch 16 bit value at main memory addr

sc wvariable tmp2 0 tmp2 cr st? cr W! cr tmp2 st? cr W@ st?

1 items cleared

ST: 0_000_000_000 0_000_017_646

ST: 0_000_017_646

ST: 0_000_000_000

Prop0 Cog6 ok

3.7.8 W!

\ W! (h1 addr --) \ store 16 bit value (h1) main memory at addr

sc variable tmp3 -1 tmp3 cr st? cr W! st? cr tmp3 st? cr W@ st?

1 items cleared

ST: -0_000_000_001 0_000_017_660

ST:

ST: 0_000_017_660

ST: 0_000_065_535

Prop0 Cog6 ok

16bit word addresses must be word aligned. Note that when there is a -1 on the stack it corresponds to h_FFFF_FFFF. When W! stores a value it only store the lower 16bits. So the value h_FFFF is stored in the word variable. When it is fetched, the value h_FFFF goes on the stack, this corresponds to d_65_535. Generally 16bit words are used unsigned values and addresses, as they take up less space than longs. If you need signed values, use 32bit longs.

3.7.9 C@

\ C@ (addr -- c1) \ fetch 8 bit value at main memory addr

sc wvariable tmp4 h_00_FF tmp4 cr st? W! st? tmp4 cr st? C@ cr st? tmp4 1 + cr st? C@ st?

1 items cleared

ST: 0_000_000_255 0_000_017_674

ST:

ST: 0_000_017_674

ST: 0_000_000_255

ST: 0_000_000_255 0_000_017_675

ST: 0_000_000_255 0_000_000_000

Prop0 Cog6 ok

When a 16bit word is stored, it is stored as 2 bytes, the lower byte is stored first, and the upper byte is stored next.

3.7.1 C!

\ C! (c1 addr --) \ store 8 bit value (c1) main memory at addr

sc wvariable tmp5 h_FF_00 tmp5 cr st? W! st? tmp5 cr st? C@ cr st? tmp5 1 + cr st? C@ st?

2 items cleared

ST: 0_000_065_280 0_000_017_686

ST:

ST: 0_000_017_686

ST: 0_000_000_000

ST: 0_000_000_000 0_000_017_687

ST: 0_000_000_000 0_000_000_255

Prop0 Cog6 ok

3.7.2 and

\ and (n1 n2 -- n1) \ bitwise and n1 n2

sc hex F0F0_FFFF AAAA_5555 cr st? cr and st? cr FF st? cr and st? decimal

2 items cleared

ST: F0F0_FFFF AAAA_5555

ST: A0A0_5555

ST: A0A0_5555 0000_00FF

ST: 0000_0055

Prop0 Cog6 ok

3.7.3 andn

\ andn (n1 n2 -- n1) \ bitwise and n1 invert n2

sc hex F0F0_FFFF AAAA_5555 cr st? cr andn st? cr FFFF st? cr andn st? decimal

1 items cleared

ST: F0F0_FFFF AAAA_5555

ST: 5050_AAAA

ST: 5050_AAAA 0000_FFFF

ST: 5050_0000

Prop0 Cog6 ok

3.7.4 invert

\ invert (n1 -- n2) bitwise invert n1

sc hex F0F0_FFFF cr st? cr invert st? decimal

1 items cleared

ST: F0F0_FFFF

ST: 0F0F_0000

Prop0 Cog6 ok

3.7.5 lshift

\ lshift (n1 n2 -- n3) \ n3 = n1 shifted left n2 bits

sc hex F000_000F 1 cr st? cr lshift st? cr 3 st? cr lshift st? decimal

1 items cleared

ST: F000_000F 0000_0001

ST: E000_001E

ST: E000_001E 0000_0003

ST: 0000_00F0

Prop0 Cog6 ok

3.7.6 rshift

\ rshift (n1 n2 -- n3) \ n3 = n1 shifted right logically n2 bits

sc hex F000_00F0 3 cr st? cr rshift st? cr 1 st? cr rshift st? decimal

1 items cleared

ST: F000_00F0 0000_0003

ST: 1E00_001E

ST: 1E00_001E 0000_0001

ST: 0F00_000F

Prop0 Cog6 ok

3.7.7 or

\ or (n1 n2 -- n1_or_n2) \ bitwise or

sc hex F000_00F0 1_0001 cr st? cr or st? cr 1_0003 st? cr or st? decimal

1 items cleared

ST: F000_00F0 0001_0001

ST: F001_00F1

ST: F001_00F1 0001_0003

ST: F001_00F3

Prop0 Cog6 ok

3.7.8 xor

\ xor (n1 n2 -- n1_xor_n2) \ bitwise xor

sc hex F000_00F0 1_0001 cr st? cr xor st? cr 1_0003 st? cr xor st? decimal

1 items cleared

ST: F000_00F0 0001_0001

ST: F001_00F1

ST: F001_00F1 0001_0003

ST: F000_00F2

Prop0 Cog6 ok

3.7.9 >

\ > (n1 n2 -- t/f) \ flag is true if and only if n1 is greater than n2

sc 1 0 st? cr > . cr cr 1 1 st? cr > . cr cr 0 -1 st? cr > . cr cr

0 items cleared

ST: 0_000_000_001 0_000_000_000

-1

ST: 0_000_000_001 0_000_000_001

0

ST: 0_000_000_000 -0_000_000_001

-1

Prop0 Cog6 ok

3.7.10 =

\ = (n1 n2 -- t/f) \ compare top 2 32 bit stack values, true if they are equal

sc 1 0 st? cr = . cr cr 1 1 st? cr = . cr cr 0 -1 st? cr = . cr cr

0 items cleared

ST: 0_000_000_001 0_000_000_000

0

ST: 0_000_000_001 0_000_000_001

-1

ST: 0_000_000_000 -0_000_000_001

0

Prop0 Cog6 ok

3.7.11 <

\ < (n1 n2 -- t/f) \ flag is true if and only if n1 is less than n2

sc 1 0 st? cr < . cr cr 1 1 st? cr < . cr cr 0 1 st? cr < . cr cr

0 items cleared

ST: 0_000_000_001 0_000_000_000

0

ST: 0_000_000_001 0_000_000_001

0

ST: 0_000_000_000 0_000_000_001

-1

Prop0 Cog6 ok

3.7.12 <>

\ <> (x1 x2 -- flag) flag is true if and only if x1 is not bit-for-bit the same as x2.

sc 1 0 st? cr <> . cr cr 1 1 st? cr <> . cr cr 0 -1 st? cr <> . cr cr

0 items cleared

ST: 0_000_000_001 0_000_000_000

-1

ST: 0_000_000_001 0_000_000_001

0

ST: 0_000_000_000 -0_000_000_001

-1

Prop0 Cog6 ok

3.8 Strings

In PropForth strings are stored as a series of bytes, where the first byte is the unsigned length of the string, and the bytes of the string immediately follow. String can be from 0 to 255 bytes long, and they consume 1 to 256 bytes of memory. To declare a string, c" string_characters" is used. The c" is followed by a space which is not part of the string, and ends with " which is not part of the string. When a string is declared at the prompt, it is only valid until that line is finished executing, as it is in temp storage. When a string is declared in a word definition, the string will be written to the dictionary, and will be valid until reboot.

3.9 Forth Words - Part 3

3.9.1 .cstr

\ .cstr (addr --) emit a counted string at addr

sc c" Hello world" cr st? cr .cstr cr st?

1 items cleared

ST: 0_000_002_194

Hello world

ST:

Prop0 Cog6 ok

3.9.2 ."

\ ." string_chars" is the equivalent of c" string_chars" .cstr in a definition

```
sc : helloworld c" Hello world" .cstr cr ." HELLO WORLD" cr ; cr helloworld

0 items cleared

Hello world

HELLO WORLD

Prop0 Cog6 ok
```

If we wish to embed non printable characters, or quotes in a string we can use ~nnn where nnn is a valid number 3 characters long. This will embed one character, with ascii value nnn in the string. Since the number is interpreted according to the current settings, is best use ~hxx where xx are 2 valid hex digits.

3.9.3 String examples

```
\ String examples

c" ~h0D~h0DCR example - Hello~h0DWorld~h0D~h0D~h0D" .cstr

c" ~h0D~h0D TAB example - Hello~h09~h09~h09World~h0D~h0D" .cstr

c" ~h0D~h0D LF example - Hello~h0AWorld~h0D" .cstr

c" ~h0D~h0DCR example - Hello~h0DWorld~h0D~h0D~h0D" .cstr

CR example - Hello

World

Prop0 Cog6 ok

c" ~h0D~h0D TAB example - Hello~h09~h09~h09World~h0D~h0D" .cstr

 TAB example - Hello                    World

Prop0 Cog6 ok
```

```
c" ~h0D~h0D LF example - Hello~h0AWorld~h0D" .cstr
```

```
 LF example - Hello

                World
```

Prop0 Cog6 ok

3.10 Control Flow

These words are only valid while defining a word. They are not valid at the prompt.

3.10.1 Introduction to if --- then

If the value on top of the stack is any value other than 0, execute the words between if and then. If the value is zero, jump to the word after then.

3.10.2 Introduction to if --- else --- then

If the value on top of the stack is any value other than 0, execute the words between if and else. If the value is zero, execute the words between else and then.

3.10.3 Introduction to begin --- until

until will check the value on top of the stack. If it is 0, it will jump back to begin, otherwise it will continue with the word following until.

3.10.4 Introduction to do --- loop

hiboundary loboudary do xxx loop. This will increment from loboundary, and execute xxx until the value reaches hiboundary -1. Between the do and loop, the word i will put the current loop counter on the top of the stack.

3.11 Forth Words - Part 4

3.11.1 if --- then

```
\ if then

sc : test1 st? if ." executing if" then cr st? cr ;  0 test1 1 test1 -1 test1
```

```
0 items cleared

ST: 0_000_000_000



ST:
```

ST: 0_000_000_001

executing if

ST:

ST: -0_000_000_001

executing if

ST:

Prop0 Cog6 ok

3.11.2 if --- else --- then

\ if else then

sc : test2 st? if ." executing if" else ." executing else" then cr st? cr ; 0 test2 1 test2 -1 test2

0 items cleared

ST: 0_000_000_000

executing else

ST:

ST: 0_000_000_001

executing if

ST:

ST: -0_000_000_001

executing if

ST:

Prop0 Cog6 ok

3.11.3 begin --- until

\ begin until

sc : test3 begin st? 1 - dup 0 = st? until ; 3 test3

1 items cleared

ST: 0_000_000_003

ST: 0_000_000_002 0_000_000_000

ST: 0_000_000_002

ST: 0_000_000_001 0_000_000_000

ST: 0_000_000_001

ST: 0_000_000_000 -0_000_000_001

3.11.4 do --- loop

\ do loop

sc : test4 do i . loop cr st? ; 4 0 st? cr test4

2 items cleared

ST: 0_000_000_004 0_000_000_000

0 1 2 3

ST:

Prop0 Cog6 ok

3.12 Talking to the other cogs

Up until this point we have been interacting with cog 6. What about the rest of the cogs?

When PropForth starts up, it starts a serial driver on cog 7, this is assumed to be connected to a terminal, or console. The console is connected to cog 6 on startup. We can easily connect other cogs to the console.

3.13 Forth Words - Part 5

3.13.1 >con

\ >con (n1 --) disconnect the current cog, and connect the console to the cog n1

sc 6 st?

5 >con

sc 5 st?

6 >con

st?

5 >con

st?

6 >con

st?

sc 6 st?

0 items cleared

ST: 0_000_000_006

Prop0 Cog6 ok

5 >con

Prop0 Cog5 ok

sc 5 st?

0 items cleared

ST: 0_000_000_005

Prop0 Cog5 ok

6 >con

Prop0 Cog6 ok

st?

ST: 0_000_000_006

Prop0 Cog6 ok

5 >con

Prop0 Cog5 ok

st?

ST: 0_000_000_005

Prop0 Cog5 ok

6 >con

Prop0 Cog6 ok

st?

ST: 0_000_000_006

Prop0 Cog6 ok

When we are connected to cog 6, and we type in 5 >con(ENTER), we connect the console to the cog 5. Because input to a cog has a buffer of one character, if we type quickly enough the first character following may be sent to the currently connected cog. So we make sure to type 2 additional enters. The first additional enter will be consumed by cog 6, and the second by cog 5. Realistically, if you are typing, this will not be an issue, but if you paste in text, or are sending it programmatically, this can occur.

3.13.2 cogx

\ cogx (cstr n --) execute cstr on cog n

sc c" 111 222 333" 0 cogx st?

0 >con

st?

6 >con

st?

0 items cleared

ST:

Prop0 Cog6 ok

0 >con

Prop0 Cog0 ok

st?

ST: 0_000_000_111 0_000_000_222 0_000_000_333

Prop0 Cog0 ok

Prop0 Cog0 ok

6 >con

Prop0 Cog6 ok

st?

ST:

Prop0 Cog6 ok

We can use another cog by sending it a command. If we know the cog is waiting at the command prompt, cogx will send it the command string. This is useful for starting up programs, and monitoring status.

3.13.3 cogreset

\ cogreset (n1 --) reset the forth cog

0 cogreset

Prop0 Cog6 ok

CON:Prop0 Cog0 RESET - last status: 0 ok

When we do not know the status of a cog, or we have set a word running on a cog, and we want it to stop, cogreset is how we do it.

3.13.4 Underflow error

\ underflow error example

c" sc + + +" 0 cogx

Prop0 Cog6 ok

CON:Prop0 Cog0 RESET - last status: 3 MAIN STACK UNDERFLOW

If we make a mistake, the cog will write an error to the console. If other cogs are trying write to the console at the same time, the messages may get a little garbled, the price of concurrency. In this example we send cog 0 a command which will cause an error, and we dump our stack 3 times, note how the error message can get interleaved with our output.

```
\ underflow error example 2

1 2 3 4 5 6 c" sc + + +" 0 cogx st? st? st?

1 2 3 4 5 6 c" sc + + +" 0 cogx st? st? st?

ST: 0_000_000_001 0_000_000_002 0_000_000_003

CON:Prop0 Cog0 RESET - l0_000_000_004 st status: 3 MAIN STACK UNDERFLOW

0_000_000_005 0_000_000_006

ST: 0_000_000_001 0_000_000_002 0_000_000_003 0_000_000_004 0_000_000_005
0_000_000_006

ST: 0_000_000_001 0_000_000_002 0_000_000_003 0_000_000_004 0_000_000_005
0_000_000_006

Prop0 Cog6 ok

Prop0 Cog6 ok
```

3.13.5 .const?

```
\ .const? ( -- ) prints out the stack to the console

c" sc 111 222 333 .const?" 0 cogx

Prop0 Cog6 ok

ST: 0_000_000_111 0_000_000_222 0_000_000_333
```

If a cog does not have the console connected, the output of st? is discarded. We can use .const?, which behaves like st? but send the output to the console. This allows us to see the stack on another cog.

4 PropForth pin IO

One of the most common tasks on a microcontroller is input from pins, and output to pins. PropForth makes this easy.

The examples we will be using, do not require any special hardware as we will be using the

capabilities of the propeller to write pins from one cog, and read them with another.

Before we move on to examples, we will have to introduce a few more words.

4.1 Forth Words - Part 5

4.1.1 space

\ space (--) emits a space

sc c" hello" .cstr space c" world" .cstr cr

0 items cleared

hello world

Prop0 Cog6 ok

4.1.2 delms

\ delms (n1 --) delay n1 milli-seconds for 80Mhz h68DB max

sc 2000 delms st?

0 items cleared

ST:

Prop0 Cog6 ok

4.1.3 pinin

\ pinin (n1 --) set pin # n1 to an input

sc 0 st? cr pinin st?

0 items cleared

ST: 0_000_000_000

ST:

Prop0 Cog6 ok

4.1.4 pinout

\ pinout (n1 --) set pin # n1 to an output

sc 0 st? cr pinout st?

0 items cleared

ST: 0_000_000_000

ST:

Prop0 Cog6 ok

4.1.5 pinlo

\ pinlo (n1 --) set pin # n1 to lo

sc 0 st? cr pinlo st?

0 items cleared

ST: 0_000_000_000

ST:

Prop0 Cog6 ok

4.1.6 pinhi

\ pinhi (n1 --) set pin # n1 to hi

sc 0 st? cr pinhi st?

0 items cleared

ST: 0_000_000_000

ST:

Prop0 Cog6 ok

4.1.7 px

\ px (t/f n1 --) set pin # n1 to high when true or low when false

sc 0 0 st? cr px st? -1 0 st? cr px st?

0 items cleared

ST: 0_000_000_000 0_000_000_000

ST:

ST: -0_000_000_001 0_000_000_000

ST:

Prop0 Cog6 ok

4.1.8 px?

\ px? (n1 -- t/f) true if pin n1 is hi

sc 0 st? cr px? st? 31 st? cr px? st?

0 items cleared

ST: 0_000_000_000

ST: 0_000_000_000

ST: 0_000_000_000 0_000_000_031

ST: 0_000_000_000 -0_000_000_001

Prop0 Cog6 ok

4.1.9 pins?

\ pins (--) samples the current status of all the io pins and prints them to the terminal

```
: pins?

  ." 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31" cr

  32 0 do i pinin i px? if ." HI " else ." LO " then loop cr

;
```

pins?

```
: pins?

  ." 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31" cr

  32 0 do i pinin i px? if ." HI " else ." LO " then loop cr

;
```

Prop0 Cog6 ok

pins?

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

LO LO LO LO LO LO LO LO LO HI HI LO HI HI HI LO LO LO LO LO LO LO LO LO LO LO LO LO HI HI HI HI

Prop0 Cog6 ok

Now having defined pins? We can start turning pins on and off and look at them.

4.1.10 IO example 1

\ IO example 1

```
c" 0 pinout 0 pinlo" 0 cogx cr cr

pins? cr cr

c" 0 pinhi" 0 cogx cr cr

pins? cr cr

c" 0 pinin" 0 cogx cr cr

pins? cr cr

100 delms pins? cr cr

c" 0 pinout 0 pinlo" 0 cogx cr cr
```

Prop0 Cog6 ok

pins? cr cr

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

LO LO LO LO LO LO LO LO LO HI HI LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO HI HI HI HI

Prop0 Cog6 ok

c" 0 pinhi" 0 cogx cr cr

Prop0 Cog6 ok

pins? cr cr

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

HI LO LO LO LO LO LO LO LO HI HI LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO HI HI HI HI

Prop0 Cog6 ok

c" 0 pinin" 0 cogx cr cr

Prop0 Cog6 ok

pins? cr cr

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

HI LO LO LO LO LO LO LO LO HI HI LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO HI HI HI HI

Prop0 Cog6 ok

100 delms pins? cr cr

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

LO LO LO LO LO LO LO LO LO HI HI LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO LO HI HI HI HI

Prop0 Cog6 ok

We are only interested in pin 00 at this point. On this system pin 00 has nothing connected to

it, so when we make it an input, it may be hi or lo. Generally after a period of time the pin will read lo. It is not a good idea to rely on this, but it is good to be aware.

4.1.11 IO example 2

```
\ IO example 2
: toggle
  0 pinout
  begin
    0 pinlo
    500 delms
    0 pinhi
    500 delms
    0
  until
;
0 cogreset c" toggle" 0 cogx
pins? cr 200 delms pins? cr 200 delms pins? cr 200 delms pins? cr 200 delms pins? cr
: toggle
  0 pinout
  begin
    0 pinlo
    500 delms
    0 pinhi
    500 delms
    0
  until
;
Prop0 Cog6 ok
```

```
0 cogreset c" toggle" 0 cogx


CON:Prop0 Cog0 RESET - last status: 0 ok

Prop0 Cog6 ok

pins? cr 200 delms pins? cr 200 delms pins? cr 200 delms pins? cr 200 delms pins? cr

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

LO LO LO LO LO LO LO LO LO HI HI LO HI HI HI LO LO LO LO LO LO LO LO LO LO LO LO
LO HI HI HI HI


00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

LO LO LO LO LO LO LO LO LO HI HI LO HI HI HI LO LO LO LO LO LO LO LO LO LO LO LO
LO HI HI HI HI


00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

LO LO LO LO LO LO LO LO LO HI HI LO HI HI HI LO LO LO LO LO LO LO LO LO LO LO LO
LO HI HI HI HI


00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

HI LO LO LO LO LO LO LO LO HI HI LO HI HI HI LO LO LO LO LO LO LO LO LO LO LO LO
LO HI HI HI HI


00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

HI LO LO LO LO LO LO LO LO HI HI LO HI HI HI LO LO LO LO LO LO LO LO LO LO LO LO
LO HI HI HI HI
```

Prop0 Cog6 ok

We set one cog modifying a signal with a cycle of one second, and we set another cog looking at the signal 5 times a second. If the signal is changing more often than we are looking at it, it is pretty obvious we are going to miss some changes. This an important concept when sampling signals, we need to look at the signal at least twice as rapidly as it is changing (search the web for Nyquist and sampling for details). If we need to see the signal edges in

relation to other signals, we will need to sample more rapidly. To start looking at this in more detail, we will need to cover LogicAnalyzer. This is a PropForth tool that looks at the pins and displays them graphically.

5 LogicAnalyzer

When we need to look at changing io pins, and understand what they are doing, Logic Analyzer that looks at the pins a user determined intervals, and store the results. It can only understand if a pin is hi or lo, if it is left as a floating input, it may appear as either, or as changing. When Logic Analyzer samples, it either uses cog memory for fast sampling or free dictionary memory for slower sampling.

Logic Analyzer has 2 flavors, a console mode, invoked by the word lac, and words which you can call directly from the command line or other words. We will cover the console mode.

To use Logic Analyzer and be able to correctly interpret what you are seeing there are a few concepts to cover.

The first is triggering, when does it start sampling. It can start sampling on the rising edge of a pin, then falling edge of a pin, or it can ignore the trigger, and sample as soon as it is ready (trigger NONE). For lac, if a pin it not changing rapidly enough, a few times a second, lac will not be able to determine the frequency of the trigger. In this case it will ignore the trigger setting and sample when it is ready.

Sample Interval is the next important item. This is what defines how often lac looks at the pins. Look too slowly, and you will miss something, look too quickly and you may not get the big picture. We will see examples of both.

The user interface of lac uses single keys to change values and start sampling. It requires an ANSI terminal, as it running in full screen. A previous version of lac had a much more elaborate interface, but it consumed too much memory, and thus did not have adequate space for the samples.

5.1 Loading lac

In the download of PropForth, in the directory CurrentRelease\Extensions, open the file lac.f and paste the contents in the terminal.

If you wish to have lac always available, type in saveforth, this will save the running image to eeprom, and lac wil be available after rebooting.

5.2 Running lac

lac

Trigger Pin -q +Q: 0

Trigger Edge -w +W SPACE: __--

Trigger Frequency eE :

Sample Interval -asdfg +ASDFG: 500.0 nS 40 clock cycles

Sample <ENTER>

Quit <ESC>

Sample Display -zxcv +ZXCV : clock cycles of 128

Starting from the top, the trigger pin is currently pin 0. Q will decrement this number and Q will increment this number. When this pin changes, depending on other settings is when we will start sampling.

The trigger edge indicates we are looking for a low to high transition on this pin, w, W and space will set this setting to --__ or --__ or NONE. When we set this to NONE, there are no specific criteria for when we start sampling.

The trigger Frequency is set when the trigger pin changes, or e or E is hit. lac will look at the trigger pin for about 250 msec to try to calculate the trigger frequency. If E is hit, it will look at the pin for a about a second to calculate the trigger frequency. If the pin is changing slower than once per second, lac may not detect the frequency. If lac does not detect the frequency, it will behave like trigger is set to NONE. This is to prevent it from waiting forever for a change which will not happen.

Sample interval is the next field. This defines how often we are going to look at the pins. lac will run in a few different modes depending on what this value. The most important item to note is that if you set it to sample at every clock cycle, it will reset cogs 2-5 and use them to do the sampling. (This parameter can be set to any 4 sequential cogs). In all other cases only the current cog is used. All the pins on the cogs used by current cog (or 2-5) are set to be inputs. This may seem obvious, but if you set a pin to output, and set it high, and then run lac, lac will change that pin to an input to sample it. The sampling interval is changed by the keys a-g and A-G. Lower case decrements the interval and upper case increments it. a and A change it by one cycle, and g-G by 10,000 cycles, with the keys in between changing it by 10, 100, and 1000.

If you want to exit lac, hit ESC. To have lac sample and display hit ENTER.

Sample Display tells which samples of the total sample you are looking at. To scroll the sample display use zxcv ZXCV.

Now for an example, we are going to define a simple program which increment pins. Enter / paste in the following and hit enter to get lac to sample.

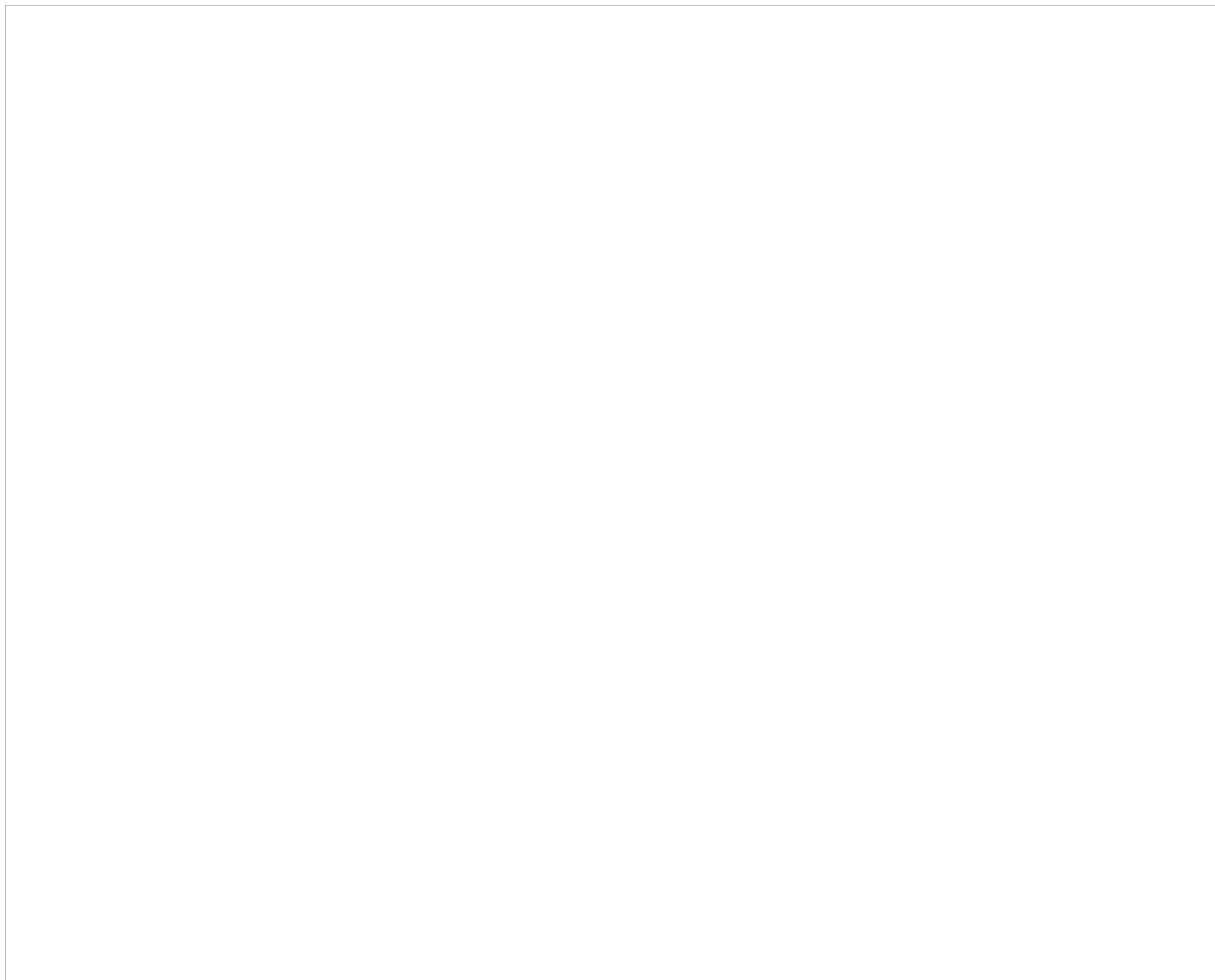
```
\ pinIncrementer( n1 ) increment 8 pins starting at pin n1
: pinIncrement
  hFF over lshift dira COG!
```

```
  0 begin
    1+ 2dup swap lshift outa COG!
  0 until
;
c" 0 pinIncrement" 0 cogx
```

lac

We should see something like the following, notice that we only see one transition, so we are not really getting a good picture of what is going on.

Change the Trigger Pin to pin 7, the Trigger Edge to --__and set the sample to 1000 clock cycles, and resample. Now we get a much better picture of what is going on.



Now if we hit C twice, we can look at the signals further along. This gives us a fair bit of detail, and still see a large enough picture.

Note that when you scroll the Sample Display, it will stay there when you resample.

Remember that lac is just looking at the pins every sample interval as specified, and then allowing us to look at them graphically. If we sample too fast or slow, we will not have a good picture of what is happening. Try to get a few samples at different rates to understand how this can affect thing.

Sampling too slow can really give us a bad picture, which can be very misleading. Here is the same signal sampled every 20,000 cycles. It look like pin 0 is changing at the same rate as pin 6. We are sampling way too slow, and we happen to be sampling at a frequency that makes pin 0 look ok, but is really very wrong. This undersampling can lead to bad information.

Undersampling is one of the most common errors leading to bad information, remember to sample and look at the signal at multiple sample rates.

6 PropForth Input/Output

6.1 Overview

A PropForth cog by default has a single character IO channel. This channel consists of a character input channel and a character output channel. The input channel is where input is received and interpreted by PropForth, and the output channel is where the interpreter echoes characters, and sends any output. This IO channel is a synchronous channel. When a cog's output channel is not connected, the output is thrown away.

In normal operation a cog's output channel is connected to another cog's input channel. This is a synchronous operation. A cog will output only as fast as the connected cog will receive characters.

When PropForth starts, cog 7 is repurposed as a serial cog. Cog 7's IO channel is buffered and received/sent to the serial driver pins. The serial driver has a circular receive buffer of 128 bytes, and a circular transmit buffer of 64 bytes. When a character is received on the serial line, it is put in the next free position in the receive buffer. If the receive buffer is full, it wraps around and starts overwriting the buffer. If the output channel is not connected, the serial driver throws away the characters received. However, if it is connected, it will synchronously send the next available character to the connected cog.

Normal startup is to connect the serial driver to cog 6. The cog? word is one way to see this.

cog?

```
Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->7(0)

Cog:7  #io chan:1                          SERIAL  7(0)->6(0)
```

Prop0 Cog6 ok

This word looks at each cog, and prints out how many IO channels the cog has, a string of what the cog is running, and its IO connections. Most times a cog will have only one IO channel. The exception is a cog running an IP server, or MCS (Multi Channel Synchronous) communications. If we look at cog 7, it tells us that 7(0) (cog 7 channel 0) -> (is connected to) 6(0) (cog 6 channel 0). This means serial driver is sending the characters it receives to cog 6. Conversely, cog 6 is sending its output to cog 7's input, which the serial driver buffers and transmits.

```
5 >con
```

```
cog?

4 >con
```

cog?

6 >con

cog?

5 >con

Prop0 Cog5 ok

Prop0 Cog5 ok

Prop0 Cog5 ok

cog?

Cog:0 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:4 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:5 #io chan:1 PropForth v5.0 2012JAN09 14:30 1 5(0)->7(0)

Cog:6 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:7 #io chan:1   SERIAL 7(0)->5(0)

Prop0 Cog5 ok

4 >con

Cog:0 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

```
Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  4(0)->7(0)

Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:7  #io chan:1                          SERIAL  7(0)->4(0)

Prop0 Cog4 ok

6 >con

Prop0 Cog6 ok

cog?

Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->7(0)

Cog:7  #io chan:1                          SERIAL  7(0)->6(0)

Prop0 Cog6 ok
```

Recall >con from the getting started section, all it is doing is disconnecting and connection IO channels. This is a very straightforward case. There are more complex uses which involve linking cogs. The most common one is fl .

```
fl



\ a gratuitous comment

                    cogid st? drop

{

A comment block

}

cog?
```

```
fl

Prop0 Cog5 ok

cogid st? drop

ST: 0_000_000_005

Prop0 Cog5 ok


Prop0 Cog5 ok

cog?

Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  5(0)->7(0)

Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->5(0)

Cog:7  #io chan:1                          SERIAL  7(0)->6(0)

Prop0 Cog5 ok


Prop0 Cog6 ok
```

fl takes the input it is receiving, strips out comments, and comment blocks, strips leading whitespace, buffers it, and simultaneously send it to another cog to process. This allows PropForth to accept input faster than it can interpret or compile it. So in this case we see cog 5 is now linked between cog 6 and cog 7. Cog 7 is the serial driver, sends the characters received to cog 6 who strips out comments, does buffering, and send the characters to cog 5, who does the actual processing of the input, and send it’s output to cog 7, who buffers and transmits over the serial line.

```
\ toupperdemo ( -- ) receive a character and echoes the upper case character

: toupperdemo
```

```
  begin
    key
    dup h_61 h_7A between
    if
      h_20 -
    then
emit
    0
  until
;

3 cogreset c" toupperdemo" 3 cogx cogid 3 iolink
cog?
: toupperdemo
  begin
    key
    dup h_61 h_7A between
    if
      h_20 -
    then
emit
    0
  until
;
Prop0 Cog6 ok
```

Prop0 Cog6 ok

3 cogreset c" toupperdemo" 3 cogx cogid 3 iolink

CON:Prop0 Cog3 RESET - last status: 0 ok

PROP0 COG6 OK

COG?

COG:0 #IO CHAN:1 PROPFORTH V5.0 2012JAN09 14:30 1

COG:1 #IO CHAN:1 PROPFORTH V5.0 2012JAN09 14:30 1

COG:2 #IO CHAN:1 PROPFORTH V5.0 2012JAN09 14:30 1

COG:3 #IO CHAN:1 PROPFORTH V5.0 2012JAN09 14:30 1 3(0)->7(0)

COG:4 #IO CHAN:1 PROPFORTH V5.0 2012JAN09 14:30 1

COG:5 #IO CHAN:1 PROPFORTH V5.0 2012JAN09 14:30 1

COG:6 #IO CHAN:1 PROPFORTH V5.0 2012JAN09 14:30 1 6(0)->3(0)

COG:7 #IO CHAN:1   SERIAL 7(0)->6(0)

PROP0 COG6 OK

Cog 3 is converting the output of cog 6 to upper case, and then forwarding it to the serial driver. Cog 6 still receive the characters as lower case and processes them as such, but when it echoes the characters or generates any output, cog 3 turns them to uppercase.

cogid iounlink 3 cogreset 100 delms cog?

COGID IOUNLINK 3 COGRESET 100 DELMS COG?

CON:Prop0 Cog3 RESET - last status: 0 ok

Cog:0 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:4 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:5 #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:6 #io chan:1 PropForth v5.0 2012JAN09 14:30 1 6(0)->7(0)

Cog:7 #io chan:1   SERIAL 7(0)->6(0)

Prop0 Cog6 ok

And back to normal.

6.2 Technical details

Each cog has 224 bytes of main memory which hold a number of variables. This cog data area has the beginning the IO channel which PropForth uses for input and output. The IO channel is 2 words, the first word is the input character to PropForth. The second word is a pointer to an input channel, or zero if the channel is not connected.

6.2.1 Input Word

When another cog wants to send input to another cogs, it waits until the value of that cog's input word is h_0100. It then writes a word with the value h_00xx where xx is the character value. The receiving cog waits until the h_0100 anded with the input word is zero, and it then accepts the character. When it is ready to accept another character, it write h_0100 to the word. This is how all the PropForth IO words read a character.

\ io (-- addr) the address of the io channel for the cog

sc io st?

0 items cleared

ST: 0_000_002_184

Prop0 Cog6 ok

Obviously this will return a different value for each cog, if we need another cog's IO channel / data area:

\ cogio (n -- addr) the address of the data area for cog n

sc 0 cogio 1 cogio 2 cogio st?

0 items cleared

ST: 0_000_000_840 0_000_001_064 0_000_001_288

Prop0 Cog6 ok

\ input demo, send the key 5 to cog 0

sc 0 cogreset 10 delms hex 0 cogio W@ st? drop h_35 0 cogio W! 0 cogio W@ st? decimal

0 items cleared

CON:Prop0 Cog0 RESET - last status: 0 ok

ST: 0000_0100

ST: 0000_0100

Prop0 Cog6 ok

Cog 0 is waiting for input, when we write a character, cog 0 accept it, and writes h_0100 back. The io routines in PropForth and optimized in assembler, so the time it takes cog 0 to accept the word, and be ready for another word is very short. So we will make cog 0 busy.

\ input demo, send the key 5 to cog 0, after we make it busy

sc 0 cogreset 10 delms hex

: busy begin 0 until ;

c" busy" 0 cogx

0 cogio W@ st? drop h_35 0 cogio W! 0 cogio W@ st?

decimal

0 items cleared

CON:Prop0 Cog0 RESET - last status: 0 ok

Prop0 Cog6 ok

: busy begin 0 until ;

Prop0 Cog6 ok

c" busy" 0 cogx

Prop0 Cog6 ok

0 cogio W@ st? drop h_35 0 cogio W! 0 cogio W@ st?

ST: 0000_0100

ST: 0000_0035

Prop0 Cog6 ok

decimal

Prop0 Cog6 ok

After we write the character we can see cog 0 has not yet accepted it, and will not since we made it busy.

6.2.2 Output Pointer

The second word of the IO channel is a pointer. If the pointer is zero, all the output to the IO channel is discarded. This prevents a cog from blocking. When a cog is connected, it will point to the input word of the channel.

```
\ cog ouput pointer example

sc cog? 5 cogio 2+ W@ 6 cogio 2+ W@ 7 cogio st?

0 items cleared

Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  6(0)->7(0)

Cog:7  #io chan:1                        SERIAL  7(0)->6(0)

ST: 0_000_000_000 0_000_002_408 0_000_002_408

Prop0 Cog6 ok
```

We can see that the output pointer of cog 5 is zero, not connected. The output pointer of cog 6 points to the input word of cog7, which is the serial driver.

```
5 >con



sc cog? 5 cogio 2+ W@ 6 cogio 2+ W@ 7 cogio st?

6 >con



5 >con

Prop0 Cog5 ok

sc cog? 5 cogio 2+ W@ 6 cogio 2+ W@ 7 cogio st?
```

```
0 items cleared

Cog:0  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:1  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:2  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:3  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:4  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:5  #io chan:1 PropForth v5.0 2012JAN09 14:30 1  5(0)->7(0)

Cog:6  #io chan:1 PropForth v5.0 2012JAN09 14:30 1

Cog:7  #io chan:1                           SERIAL  7(0)->5(0)

ST: 0_000_002_408 0_000_000_000 0_000_002_408

Prop0 Cog5 ok

6 >con

Prop0 Cog6 ok
```

Now see that the output pointer of cog 5 points to the input word of cog7, which is the serial driver. And the output pointer of cog 6 is zero.

6.2.3 Advanced topics

Each PropForth cog has a single IO channel. A serial driver running on a cog has one IO channel as well. This is channel zero, and in most cases the only channel. There are cases where a cog is running a driver that has more than one channel. Two examples of this are the MCS driver, and the IP server.

6.2.3.1 MCS (Multi Channel Synchronous) Driver

MCS is a driver that uses 2 pins to provide 8 full duplex IO channels to another propeller running MCS. In this case the cog running MCS has an array of 8 IO channels, starting where the normal IO channel resides. This means that when we want to address an MCS channel, we need to specify the cog running MCS, and which channel we wish to address. The native PropForth words which require both the cog and the channel to be specified (ioconn) (iodis) (iolink) (iounlink). The words ioconn, iodis, iolink, iounlink calls these words with a channel number of zero.

For further details see the MCS documentation.

6.2.3.2 IP Server

The IP server interfaces to the WIZNET W5100 to provide 4 Internet Protocol channels. Like any channel, these can connect to cogs and provide IO. In the case when all the channels are started as telnet channels, each channel can connect to a cog. This allows talking to 4 cogs at

the same time with 4 telnet sessions. Very handy for some applications.

For further details see the IP documentation.

7 PropForth Assembler

The propeller has a unique architecture and PropForth was written to match the architecture. Using assembler on the propeller is not the same as using assembler on a more conventional architecture, and it is very different than a CISC architecture like what you find in your PC. Firstly each cog has 496 registers that are accessible only to that cog for instructions and data. Each cog can rely on the fact that only it can modify those registers, short of the cog being reset externally. In addition, each cog will never be interrupted, it will continually execute and it can decide when to access main memory or not. This can give each cog a high degree of isolation from the other cogs.

PropForth is an easy way to explore the propeller assembler, short words can be defined, and executed and the results can be easily integrated into normal PropForth words, or they can be used interactively via the console. While this does not change the complexity of assembler code, it does provide an easy interface to that code.

One of the changes in PropForth v5.0 was the way in which assembler words are defined and used. An assembler word is defined, and can be invoked just like any other forth word. The assembler code is automatically paged in to cog memory and executed. This allowed the definition of many more assembler words, without using more cog memory.

The result was many more words defined as assembler words, and the overall kernel speed was greatly improved, and total cog memory usage went down.

To understand how to use assembler words, it is necessary to understand how they fit into the kernel.

7.1 Kernels and Assembler Words

PropForth starts life as PropForthStartKernel. This kernel has no assembler words defined, has all the symbols defined, is slow to interpret or compile anything, has error detection, and no error reporting. The cog memory usage is small. It uses a total of 274 cog longs, and 13,594 main memory bytes. The cog memory usage includes:

- 32 longs for the main stack
- 32 longs for the return stack
- 6 longs for temporary registers tregone - tregsix
- 5 longs for constants, fDestInc, fCondMask, fMask, fAddrMask, fLongMask

- 1 long for the a register used for reset, resetDreg
- 1 long used for the interpreter instruction pointer, IP
- 1 long for the stack pointer, stPtr
- 1 long for the return stack pointer, rsPtr
- 1 long for the top of stack value, stTOS
- 194 longs for the forth interpreter

This leaves a total of 222 cog longs free (512 -16 (special purpose regs) - 274 (PropForth regs). The only use for this kernel is to build other kernels. The first useful kernel, is PropForthBootKernel. This kernel removes the symbols which are necessary for rebuilding the start kernel, but are used infrequently. They can be dynamically defined as needed. The PropForthBootKernel uses 274 cog longs and 12646 main memory bytes. This kernel is still slow to interpret or compile code, but is fully functional. It is used as the base for PropForthOptimizeBootKernel. We start with PropForthBootKernel, and define a number of assembler words which will greatly speed up the kernel. Words like _accept, which reads an input line from the console, _dictsearch, which searches the dictionary, and a number of other words.

All the words are assembled to load at cog address 274, the initial value of the wvariable coghere which the first free long in cog memory.

When a word is assembled, the first long in the assembler definition is constructed as follows:

- Bits 31 - 19 (14 bits) - correspond to the 14 hi bits in main memory where the assembler word is defined, since it is always long aligned the 2 low bits will always be 0
- Bits 18 - 9 (9 bits) - the number of longs in this assembler words
- Bits 8 - 0 (9 bits) - the starting cog address of this assembler word

When the PropForth interpreter encounters an assembler word, it checks the first word of the definition against the starting cog address of the assembler word, if it matches, the assembler code for this word is already loaded, and the assembler code is executed. If it does not match, it loads the code from main memory, and then executes the assembler code. This simple mechanism allows PropForth to page in and cache assembler words into cog memory.

The kernel build process, looks at all the assembler words to and generates a word, init_coghere, which defines the new free cog starting address. This means the area between cog address 274, and 320 is used as the area where the kernel loads assembler words . This leaves room for the largest assembler word defined in the boot optimization process (_accept).

The new kernel, PropForthOptimizeBootKernel incorporates the assembler optimizations. This kernel uses 320 cog longs and 13,262 byte of main memory. If is fast to interpret and compile, but does not have any error reporting. This is added and the release kernel(s) are generated.

The release kernels do not consume more cog memory but they do consume more main memory. The starting address for the area where the assembler words are paged is defined by the constant build_BootOpt. The end of the page area is defined by the initial value of the wvariable coghere, which is 320.

Any assembler word which is to be paged in and out of this portion of cog memory, must then be defined to start at the address build_bootOpt, and be less than 46 longs. If the assembler word never returns, like _serial, which starts a serial driver on the cog, it can be longer than 46 longs, as nothing else will ever have the opportunity to be paged in.

If nothing else is using the cog memory, and the page area is overflowed, this will be a bug that doesn't show up, until something uses the free cog memory, like the sd card forth code.

So the rules for assembler words; the start address if defined by build_BootOpt, and the end address must be before the initial value of coghere. If these rules are followed, there is no danger of a problem.

LogicAnalyzer breaks these rules, the faster sampling routines (sampling faster than every 41 cycles) uses the cog memory to store samples, so if you use LogicAnalyzer on the same cogs as the sd card driver, there will be strange effects. The original LogicAnalyzer was written before these rules were established. All of the other optimizations in the sd card driver or the ipserver follow the rules with no problems.

Some programmers may quickly deduce that this mechanism can allow for multiple page areas in each cog. This capability can allow for multiple cache sets in each cogs memory. This capability has not yet been explored by the PropForth kernel.

7.2 Writing Assembler Words

The interactive nature of PropForth guides us when to use assembler. Generally the development cycle it to write forth words, verify the development functionality, and then rewrite some of the forth words in assembler to get the performance we need or want, and then of course re-verify the functionality. This has been the development cycle for the kernel and extensions.

To illustrate how to use the assembler we will generate some simple artificial examples, which illustrate the concepts. To use the assembler, load asm.f from the CurrentRelease/Extensions directory, and if you like do a saveforth.

A PropForth assembler word is always in the form:

```
build_BootOpt :rasm

   jexit

;asm noop



lockdict create noop forthentry

$C_a_lxasm w, h114  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,
```

z1SV01X l,

freedict

Prop0 Cog6 ok

In the input, build_BootOpt is the constant which defines the starting address to load this word, :rasm starts the assembler definition, jexit is an assembler macro (more on macros later) which returns control to the PropForth interpreter, and ;asm noop ends the definition and names the word noop .

The assembler does not define the word, it emits the definition which is used to define the word. The first line locks the dictionary and defines the dictionary entry, adds the forth interpreter code for assembler definitions, and generates the first long of the assembler definition. Recall, the first long defines the address in main memory where the definition resides, the starting address in cog memory, and the number of longs in the definition.

Then follow the longs which are the assembler code. These are output as base 64 numbers, this is the most compact representation. If you really want to see hex change the line:

h7A emit base W@ h40 base W! swap u. base W!

to:

h68 emit base W@ h10 base W! swap u. base W!

in asm.f

And the last line frees the dictionary. Since multiple cogs can update the dictionary, the definition must start and end with no other cogs able to update the dictionary. This is accomplished by lockdict / freedict.

Now we can take the assembler definition and paste it into the console, and run the word.

lockdict create noop forthentry

$C_a_lxasm w, h114 h113 1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

z1SV01X l,

freedict

lockdict create noop forthentry

Prop0 Cog6 ok

$C_a_lxasm w, h114 h113 1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

Prop0 Cog6 ok

z1SV01X l,

```
Prop0 Cog6 ok

freedict

Prop0 Cog6 ok
```

noop

Prop0 Cog6 ok

Not a functionally exciting example, but we have generates an assembler word which can be automatically paged in and out as needed.

The assembler generates a definition which we can load, without having the assembler present. This is great, but it will get a little tedious to have to

7.3 The Stack in Detail

To write useful assembler words, we need to understand the stack and how to manipulate it with assembler. In the execution of an assembler word, there are six temporary registers available for use. The contents of these registers is undefined every time the word is called. Some of these registers are commoly used in stack manipulations, and they are referred to as $C_treg1 - $C_treg6 .

The $C_ prefix indicated this refers to an address in cog memory, and the forth kernel rebuild procedure updates these as necessary when kernels are rebuilt.

The value on the top of the stack is held in a register called $C_stTOS . The first example we will generate is a very simple operation which changes the value on the top of the stack.

```
build_BootOpt :rasm

     add    $C_stTOS , # d_256

    jexit
```

;asm add256

```
lockdict create add256 forthentry

$C_a_lxasm w, h115  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

z20yPS0 l, z1SV01X l,

freedict
```

Prop0 Cog6 ok

All assembler mnemonics are lower case and are delimited by spaces, the next field is the destination register also delimited by spaces, and the next field is the source register, also delimited by spaces. The # indicates the following value is an immediate value, as opposed to the register location. Immediate values in assembler instructions are numbers between 0 and 511 inclusively. Any numbers used in an assembler definition should explicitly define the base as decimal, hex, binary or base64.

Now we define the example:

lockdict create add256 forthentry

$C_a_lxasm w, h115 h113 1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

z20yPS0 l, z1SV01X l,

freedict

lockdict create add256 forthentry

Prop0 Cog6 ok

$C_a_lxasm w, h115 h113 1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

Prop0 Cog6 ok

z20yPS0 l, z1SV01X l,

Prop0 Cog6 ok

freedict

Prop0 Cog6 ok

And run some tests:

sc 0 st? add256 st? add256 st? -100 add256 st?

0 items cleared

ST: 0_000_000_000

ST: 0_000_000_256

ST: 0_000_000_512

ST: 0_000_000_512 0_000_000_156

Prop0 Cog6 ok

Now we will deal with multiple items on the stack, but first the assembler macros. Assembler macros are defined in asm.f, and there are a few defined which are used repeatedly, we have

already used one, which is jexit. Macros are just shorthand for an assembler instruction with specific source and destination.

- **jexit - jmp $C_a_exit** - jumps to the PropForth interpreter assembler code which is executed when an assembler word in finished execution

- **spush - jmpret $C_a_stpush_ret , $C_a_stpush** - calls a subroutine which pushes the register $C_a_stTOS onto the stack, after which $C_a_stTOS must be set to a new value. This subroutine will cause a reset error if the stack is overflowed.

- **spopt - jmpret $C_a_stpoptreg_ret , $C_a_stpoptreg** - moves the register $C_a_stTOS into $C_a_treg1, and pops on item off the stack into $C_a_stTOS. This subroutine will cause a reset error if the stack is underflowed.

- **rpush - jmpret $C_a_rspush_ret , $C_a_rspush** - pushes the value in $C_treg5 onto the return stack. This subroutine will cause a reset error if the return stack is overflowed.

- **rspop - jmpret $C_a_rspop_ret , $C_a_rspop** - pops a value from the return stack into $C_treg5. This subroutine will cause a reset error if the return stack is underflowed.

Normally the spush, spop, spopt, and jexit are the only macros used. The return stack is where the interpreter stores the addresses of the words it needs to execute when the current word is done. If they are tampered with, havoc will result. It is useful to be able to put things on the return stack, and remove them, but care should be taken, just as when using the forth r> and >r words. Now we will define an example that uses the top two stack items.

```
\ cmpsubtest ( n1 n2 - n3) if n1 is greater than or equal to n2, n3 = n1 - n2, otherwise n3 = n1
build_BootOpt :rasm

  spopt

  cmpsub   $C_stTOS , $C_treg1 wr

  jexit

;asm cmpsubtest


lockdict create cmpsubtest forthentry

$C_a_lxasm w, h116  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

z1SyLIZ l, z3WiPRC l, z1SV01X l,
```

freedict

Prop0 Cog6 ok

The first instruction, spopt, pops a value off the stack (n2), and puts it in $C_treg1. The value in $C_a_stTOS is now n1. The assembler instruction, cmpsub, subtracts the source value from the destination value if the source is greater than or equal to the destination. Note the use of wr, to explicitly write the result. When the assembler was first written, the reference used was the Propeller Manual version 1.01. In that reference the cmpsub instruction did not write the result by default.

The assembler supports, wr, nr, wz, and wc as instruction postfix operators.

To prevent the result from being written in the destination register, use nr, to write the result in the destination register, use wr, to write the z-flag ,use wz, and to write the c-flag, use wc.

Now we define the example:

```
lockdict create cmpsubtest forthentry

$C_a_lxasm w, h116  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

z1SyLIZ l, z3WiPRC l, z1SV01X l,
```

freedict

```
lockdict create cmpsubtest forthentry

Prop0 Cog6 ok

$C_a_lxasm w, h116  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

Prop0 Cog6 ok

z1SyLIZ l, z3WiPRC l, z1SV01X l,

Prop0 Cog6 ok

freedict

Prop0 Cog6 ok
```

And run some tests:

sc 10 9 cmpsubtest st? 10 11 cmpsubtest st?

```
0 items cleared

ST: 0_000_000_001
```

ST: 0_000_000_001 0_000_000_010

Prop0 Cog6 ok

One more stack example,

```
\ hibitset? ( n1 - n1 n2) n2 is the highest bit set in n1, -1 if there are not bits set
fl


build_BootOpt :rasm

\

\ $C_treg3 is a loop counter

\ $C_treg2 is the return value, which we initialize to -1

\ $C_treg1 is a mask with one bit on, to test n1

\

mov    $C_treg3 , # d32

mov    $C_treg1 , __valhi

mov    $C_treg2 , __minusone


__loop

\

\ if the bit is NOT on in n1, the z-flag will be true (1)

\

        test    $C_stTOS , $C_treg1 wz

\

\ if the zflag was NOT set, it means we have a 1 bit in n1

\ set the return value to the loop counter - 1

\

if_nz   mov     $C_treg2 , $C_treg3

if_nz   sub     $C_treg2 , # 1
```

```
\
\ if the zflag was set, it means we have a 0 bit in n1
\ shift the mask register right, and loop
\
if_z    shr     $C_treg1 , # 1
if_z    djnz    $C_treg3 , # __loop
\
\ push n1 down on the stack
\
spush
\
\ and set the new top of stack to the return value
\
mov    $C_stTOS , $C_treg2
jexit


\
\ the variables which we will use
\
__valhi
h_8000_0000
__minusone
-1
```

;asm hibitset?

```
lockdict create hibitset? forthentry

$C_a_lxasm w, h120  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

z2WyPjW l, z2WiP[U l, z2WiPfV l, z1YFPRC l, z2W\PeE l, z24oPb1 l, zbtPW1 l, z3[tPnM l,

z1SyJQL l, z2WiPRD l, z1SV01X l, z200000 l, z3yyyyy l,

freedict

Prop0 Cog5 ok
```

Prop0 Cog6 ok

In assembler words, labels start with __ . A long can be initialized with a value as shown in __valhi and __minusone. All the prefix operators can be used in the assembler as well.

Now to define the example:

```
fl

lockdict create hibitset? forthentry

$C_a_lxasm w, h120  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

z2WyPjW l, z2WiP[U l, z2WiPfV l, z1YFPRC l, zbyPW1 l, z2W\PeE l, z24oPb1 l, z3[tPnM l,

z1SyJQL l, z2WiPRD l, z1SV01X l, z200000 l, z3yyyyy l,
```

freedict

```
fl

Prop0 Cog5 ok

lockdict create hibitset? forthentry

Prop0 Cog5 ok

$C_a_lxasm w, h120  h113  1- tuck - h9 lshift or here W@ alignl h10 lshift or l,

Prop0 Cog5 ok

z2WyPjW l, z2WiP[U l, z2WiPfV l, z1YFPRC l, zbyPW1 l, z2W\PeE l, z24oPb1 l, z3[tPnM l,
```

Prop0 Cog5 ok

z1SyJQL l, z2WiPRD l, z1SV01X l, z200000 l, z3yyyyy l,

Prop0 Cog5 ok

freedict

Prop0 Cog5 ok

Prop0 Cog5 ok

Prop0 Cog6 ok

And run some tests:

sc -1 hibitset? st? 2drop 2 hibitset? st? 2drop 1 hibitset? st? 2drop 0 hibitset? st?

0 items cleared

ST: -0_000_000_001 0_000_000_031

ST: 0_000_000_002 0_000_000_001

ST: 0_000_000_001 0_000_000_000

ST: 0_000_000_000 -0_000_000_001

7.4 Assembler Mnemonic List

These are the mnemonics the assembler recognizes. For a detailed description see the Propeller manual. The assembler is based on V1.01. There are some changes in later versions.

- abs
- absneg
- add
- addabs
- adds
- addsx
- addx
- and

- andn
- cogid
- coginit
- cogstop
- clkset
- cmp
- cmps
- cmpsub
- cmpsx
- cmpx
- djnz
- jmp
- jmpret
- lockclr
- locknew
- lockret
- lockset
- long
- max
- maxs
- min
- mins
- mov
- movd

- movi
- movs
- muxc
- muxnc
- muxnz
- muxz
- neg
- negc
- negnc
- negnz
- negz
- or
- rdbyte
- rdlong
- rdword
- rcl
- rcr
- rev
- rol
- ror
- sar
- shl
- shr
- sub

- subabs
- subs
- subsx
- subx
- sumc
- sumnc
- sumnz
- sumz
- test
- tjnz
- tjz
- waitcnt
- waitpeq
- waitpne
- waitvid
- wrbyte
- wrlong
- wrword
- xor

7.5 Assembler Prefix List

These are the prefixes the assembler recognizes. For a detailed description see the Propeller manual.

- if_always
- if_never
- if_e

- if_ne
- if_a
- if_b
- if_ae
- if_be
- if_c
- if_nc
- if_z
- if_nz
- if_c_eq_z
- if_c_ne_z
- if_c_and_z
- if_c_and_nz
- if_nc_and_z
- if_nc_and_nz
- if_c_or_z
- if_c_or_nz
- if_nc_or_z
- if_nc_or_nz
- if_z_eq_c
- if_z_ne_c
- if_z_and_c
- if_z_and_nc
- if_nz_and_c

- if_nz_and_nc
- if_z_or_c
- if_z_or_nc
- if_nz_or_c
- if_nz_or_nc

7.6 Assembler Postfix List

These are the postfixes the assembler recognizes. For a detailed description see the Propeller manual.

- nr
- wr
- wc
- wz

---

8 A Low Power example

We are going to see how we can make a propeller go into very low power mode, and then boot up when a key is hit.

We are low to accomplish this by stopping all the cogs but one, and have it wait for a transition on the serial input line.

```
fl

\ sleep ( -- ) go into low power mode until a transition is detected on the serial input line to the cog

: sleep

\ print out a message, and pause, this will allow the serial driver to output the message before we shut it down

 ." ~h0D~h0DGOING TO SLEEP~h0D~h0D" 100 delms

\ shut down all the cogs but this one
```

```
  do
    i cogid <>
    if
      i cogstop
    then
  loop
\ make sure the serial input pin is an input
  $S_rxpin pinin
\ n1 0 hubopr drop is the same as a clkset, which writes to the propeller CLK register
\ setting the propeller CLK register to zero, sets the main clock to RCSLOW (13 - 33Khz)
  0 0 hubopr drop
\ wait for the serial line to be high
  $S_rxpin >m dup waitpeq
\ wait for the serial line to be lo
  $S_rxpin >m dup waitpne
\ turn on the oscillator and PLL, assumes we have a crystal/resonator of 4 - 16 Mhz
  h_68 0 hubopr drop
\ wait at least 10 millisec for the oscillator and pll to stabilize
\ we are still running with a main clock of 13 - 33Khz, so this loop is longer than 10 ms
  100 0
  do
  loop
\ set the clock to be 16x the crystal
  h_6F 0 hubopr drop
\ and execute the onboot word, this performs the initialization sequence
  0 onboot drop
  ." ~h0D~h0D AWAKE~h0D~h0D" 100 delms
;
fl
```

```
." ~h0D~h0DGOING TO SLEEP~h0D~h0D" 100 delms

8 0

do

i cogid <>

if

i cogstop

then

loop

$S_rxpin pinin

0 0 hubopr drop

$S_rxpin >m dup waitpeq

$S_rxpin >m dup waitpne

h_68 0 hubopr drop

100 0

do

loop

h_6F 0 hubopr drop

0 onboot drop

." ~h0D~h0D AWAKE~h0D~h0D" 100 delms

;

Prop0 Cog5 ok
```

Prop0 Cog6 ok

\ now put the prop to sleep

sleep

GOING TO SLEEP

The prop is now in low power mode, until we hit a key.

CON:Prop0 Cog0 RESET - last status: 0 ok

CON:Prop0 Cog1 RESET - last status: 0 ok

CON:Prop0 Cog2 RESET - last status: 0 ok

CON:Prop0 Cog3 RESET - last status: 0 ok

CON:Prop0 Cog4 RESET - last status: 0 ok

AWAKE

CON:Prop0 Cog5 RESET - last status: 0 ok

Prop0 Cog6 ok

According to the specs, typical current consumption should be in the 25 micro-amp range. Will measure and update sometime.

9 Forth Word Reference

9.1 Core Word Set (build_BootKernel, build_BootOpt)

9.1.1

\ # (n1 -- n2) divide n1 by base and convert the remainder to a char and append to the output

9.1.2 #>

\ #> (n1 -- caddr) address of a counted string representing the output, NOT ANSI

9.1.3 #s

\ #s (n1 -- 0) execute # until the remainder is 0

9.1.4 $C_IP

\ The forth instruction pointer

9.1.5 $C_a_(+loop)

\ This word constant is an assembler addresses

9.1.6 $C_a_(loop)

\ This word constant is an assembler addresses

9.1.7 $C_a_0branch

\ This word constant is an assembler addresses

9.1.8 $C_a_2>r

\ This word constant is an assembler addresses

9.1.9 $C_a_branch

\ This word constant is an assembler addresses

9.1.10 $C_a_doconw

\ This word constant is an assembler addresses

9.1.11 $C_a_dovarw

\ This word constant is an assembler addresses

9.1.12 $C_a_exit

\ This word constant is an assembler addresses

9.1.13 $C_a_litl

\ This word constant is an assembler addresses

9.1.14 $C_a_litw

\ This word constant is an assembler addresses

9.1.15 $C_a_lxasm

\ This word constant is an assembler addresses

9.1.16 $C_a_next

\ This word constant is an assembler addresses

9.1.17 $C_fMask

\ This word constant is an assembler addresses

9.1.18 $C_resetDreg

\ This word constant is an assembler addresses

9.1.19 $C_varEnd

\ This word constant is an assembler addresses

9.1.20 $H_cogdata

\ This is a pointer to the main cogdata area

9.1.21 $H_cq

\ This is ' cq - the routine which handles the word c"

9.1.22 $H_dq

\ This is ' dq - the routine which handles the word ."

9.1.23 $H_entry

\ This is the address of the assembler which is loaded to a PropForth cog

9.1.24 $S_baud

\ Word constant, the initial starting baud rate of the serial driver

9.1.25 $S_cdsz

\ The size of the cog's data area, this will be initialized by $S_cdsz defined as a spin constant

9.1.26 $S_con

\ prop 7 is normally the console channel, this is the prop which handles communication to the console, and

\ provides the interface to the rest of the cogs

9.1.27 $S_rxpin

\ Word constant, the rxpin of the serial driver

9.1.28 $S_txpin

\ Word constant, the txpin of the serial driver

9.1.29 '

\ ' (-- addr) returns the execution token for the next name, if not found it returns 0

9.1.30 (+loop)

\ (+loop) (n1 --) \ add n1 to loop counter, branch if count is below limit, offset follows,

\ -2 is to itself, +2 is next word

9.1.31 (createbegin)

\ Internal word

9.1.32 (createend)

\ Internal word

9.1.33 (fl)

\ (fl) (-- n1) buffer input and emit n1 is the number of characters overflowed

\

\ t0 - the end of the buffer

\ t1 - the number of characters overflowed

\ fl_in - pointer to next character for input

\ dictend - pointer to the next character for output

\ initialize

9.1.34 (flout)

\ (flout) (--) attempt to output a character

9.1.35 (ioconn)

\ (ioconn) (n1 n2 n3 n4 --) connect cog n1 channel n2 to cog n3 channel n4, disconnect them from other cogs first

9.1.36 (iodis)

\ (iodis) (n1 n2 --) cog n1 channel n2 disconnect, disconnect this cog and the cog it is connected to

9.1.37 (iolink)

\ (iolink) (n1 n2 n3 n4 --) links the 2 channels, output of cog n1 channel n2 -> input of cog n3 channel n4,

\ output of n3 channel n4 -> old output of n1 channel n2

9.1.38 (iounlink)

\ (iounlink) (n1 n2 --) unlinks cog n1 channel n2

9.1.39 (loop)

\ (loop) (--) \ add 1 to loop counter, branch if count is below limit offset follows,

\ -2 is to itself, +2 is next word

9.1.40 (nfcog)

\ (nfcog) (-- n1 n2) n1 the next valid free forth cog, n2 is 0 if the cog is valid

9.1.41 (prop)

\ The default prop string

9.1.42 (version)

\ The default version string

9.1.43 +

\ + (n1 n2 -- n1+n2) \ sum of n1 & n2

9.1.44 +loop

****

9.1.45 -

\ - (n1 n2 -- n1-n2) \ subtracts n2 from n1

9.1.46 -1

\ -1 or true, used frequently

9.1.47 .

\ . (n1 --) prints the signed number on the top of the stack

9.1.48 ."

\

9.1.49 ...

\ ... followed by a CR indicates the end of file by the file system write/update word, nop at the prompt

9.1.50 .cstr

\ .cstr (addr --) emit a counted string at addr

9.1.51 .str

\ .str (c-addr u1 --) emit u1 characters at c-addr

9.1.52 .strname

\ .strname (c-addr --) c-addr points to a forth name field, print the name

9.1.53 0

\ Word constant, 0 or false, used frequently

9.1.54 0<

\ 0< (n1 -- t/f) true if n1 < 0

9.1.55 0<>

\ 0<> (n1 -- t/f) true if n1 is not zero

9.1.56 0=

\ 0= (n1 -- t/f) true if n1 is zero

9.1.57 0>

\ 0> (n1 -- t/f) true if n1 > 0

9.1.58 0>=

\ 0>= (n1 -- t/f) true if n1 >= 0

9.1.59 0branch

\ 0branch (t/f --) \ branch it top of stack value is zero 16 bit branch offset follows,

\ -2 is to itself, +2 is next word

9.1.60 1

\ Word constant, 1, used frequently

9.1.61 1+

\ 1+ (n1 -- n1+1)

9.1.62 1-

\ 1- (n1 -- n1-1)

9.1.63 2

\ Word constant, 2, used frequently

9.1.64 2+

\ 2+ (n1 -- n1+2)

9.1.65 2-

\ 2- (n1 -- n1-2)

9.1.66 2/

\ 2/ (n1 -- n1>>1) n1 is shifted arithmetically right 1 bit

9.1.67 2>r

\ 2>r (n1 n2 --) \ pop top 2 stack to RS

9.1.68 2drop

\ 2drop (n1 n2 --) drop top 2 items on the stack

9.1.69 2dup

\ 2dup (n1 n2 -- n1 n2 n1 n2) copy top 2 items on the stack

9.1.70 2lock

\

9.1.71 2unlock

\

9.1.72 3drop

\ 3drop (n1 n2 n3 --) drop top 3 items on the stack

9.1.73 4*

\ 4* (n1 -- n1<<2) n1 is shifted logically left 2 bits

9.1.74 4+

\ 4+ (n1 -- n1+4)

9.1.75 :

\

9.1.76 ;

\

9.1.77 <

\ < (n1 n2 -- t/f) \ flag is true if and only if n1 is less than n2

9.1.78 <#

\ <# (--) initialize the output area

9.1.79 <=

\ <= (n1 n2 -- t/f) true if n1 <= n2

9.1.80 <>

\ <> (x1 x2 -- flag) flag is true if and only if x1 is not bit-for-bit the same as x2.

9.1.81 =

\ = (n1 n2 -- t/f) \ compare top 2 32 bit stack values, true if they are equal

9.1.82 >

\ > (n1 n2 -- t/f) \ flag is true if and only if n1 is greater than n2

9.1.83 >=

\ >= (n1 n2 -- t/f) true if n1 >= n2

9.1.84 >con

\ >con (n1 --) disconnect the current cog, and connect the console to the cog n1

9.1.85 >in

\ >in (-- addr) access as a word, addr is the var the offset in characters from the start of the input buffer to

9.1.86 >m

\ >m (n1 -- n2) produce a 1 bit mask n2 for position n1

9.1.87 >out

\ >out (-- addr) access as a word, the offset to the current output byte

9.1.88 >r

\ >r (n1 --) \ pop stack top to RS

9.1.89 C!

\ C! (c1 addr --) \ store 8 bit value (c1) main memory at addr

9.1.90 C@

\ C@ (addr -- c1) \ fetch 8 bit value at main memory addr

9.1.91 C@++

\ C@++ (c-addr -- c-addr+1 c1) fetch the character and increment the address

9.1.92 COG!

\ COG! (n1 addr --) \ store 32 bit value (n1) at cog addr

9.1.93 COG@

\ COG@ (addr -- n1) \ fetch 32 bit value at cog addr

9.1.94 ERR

\ ERR (n1 --) clear the input queue, set the error n1 and reset this cog

9.1.95 L!

\ L! (n1 addr --) \ store 32 bit value (n1) at main memory addr

9.1.96 L@

\ L@ (addr -- n1) \ fetch 32 bit value at main memory addr

9.1.97 RS!

\ RS! (n1 n2 --) \ store n1 at the n2th position on the return stack, 0 is the top of stack

9.1.98 RS@

\ RS@ (addr -- n1) \ fetch n1th value down the return stack, 0 is the top of stack

9.1.99 ST!

\ ST! (n1 n2 --) \ store n1 at the n2th position on the stack, 0 is the top of stack

9.1.100 ST@

\ ST@ (addr -- n1) \ fetch n1th value down the stack, 0 is the top of stack

9.1.101 W!

\ W! (h1 addr --) \ store 16 bit value (h1) main memory at addr

9.1.102 W+!

\ W+! (n1 addr --) add n1 to the word contents of address

9.1.103 W@

\ W@ (addr -- h1) \ fetch 16 bit value at main memory addr

9.1.104 [if

\ [if xxx   (flag --) if flag is 0, drop all characters until], [if should be the first only only chars on the line

9.1.105 [ifdef

\ [ifdef xxx   (--) if xxx is not defined drop all characters until], [ifdef xxx should be the first only only chars on the line

9.1.106 [ifndef

\ [ifndef xxx   (--) if xxx is defined drop all characters until], [ifndef xxx should be the first only only chars on the line

9.1.107 \

\ \ (--) moves the parse pointer >in to the end of the line

9.1.108]

\

9.1.109 _accept

\ _accept (-- +n2) collect padsize -2 characters or until eol, convert ctl chars to space,

\ pad with 1 space at start & end. For parsing ease, and for the length byte when we make cstrs

9.1.110 _asmpfa>nfa

\ _asmpfa>nfa (addr -- addr) pfa>nfa for an asm word

9.1.111 _cnip

\ _ cnip (--) Use in the _xasm*>* words to get rid of litw word

9.1.112 _dictsearch

\ _dictsearch (nfa cstr -- n1) nfa - addr to start searching in the dictionary, cstr - the counted string to find

\ n1 - -1 if found, 0 if not found, a fast assembler routine

9.1.113 _dl

\ _dl(c1 --) drop lines unitl c1 is received as the first or second character in a line, a . is emitted for each line

9.1.114 _ecs

\ _ecs (--) emit a colon followed by a space

9.1.115 _eeread

\ _eeread (t/f -- c1) flag should be true is this is the last read

9.1.116 _eewrite

\ _eewrite (c1 -- t/f) write c1 to the eeprom, true if there was an error

9.1.117 _femit?

\ _femit? (c1 ioaddr -- t/f) true if the output emitted a char, a fast non blocking emit

9.1.118 _finit

\ This word variable is 0 (spin code) when the propeller is rebooted and set to non-zero when

\ forth is initialized

9.1.119 _fkey?

\ _fkey? (ioaddr -- c1 t/f) fast nonblocking key routine, true if c1 is a valid key

9.1.120 _forthpfa>nfa

\ _forthpfa>nfa (addr -- addr) pfa>nfa for a forth word

9.1.121 _if

\ _if xxx (flag --) if flag is 0, drop all characters until], [if should be the first only only chars on the line

9.1.122 _lc

\ _lc (-- addr) the address of the last character in the pad, filled by the parse word

9.1.123 _lockarray

\ _lockarray - 8 character array used to keep track of locks

\

\ \ one byte for each lock

\ \ hi 4 bits is the lockcount

\ \ lo 4 bits is the cogid

9.1.124 _maskin

\ _maskin (n -- t/f) n is the bit mask to read in

9.1.125 _maskouthi

\ _maskouthi (n --) set the bits in n hi

9.1.126 _maskoutlo

\ _maskoutlo (n --) set the bits in n low

9.1.127 _mmcs

\ _ mmcs (--) print MISMATCHED CONTROL STRUCTURE(S), then clear input keys

9.1.128 _p+

\ _p+ (offset -- addr) the offset is added to the contents of the par register, giving an address references

\ the cogdata

9.1.129 _p?

\ \ _p? (-- t/f) true if prompts and errors are on

9.1.130 _qp

\ _qp (-- cstr) we are past the open " in a string, parse the string in the pad and return a cstr

9.1.131 _serial

\ _serial (n1 n2 n3 --)

\ n1 - tx pin

\ n2 - rx pin

\ n3 - clocks/bit

\

\ h00 - h04 -- io channel

\ h04 - h84 -- the receive buffer

\ h84 - hC4 -- the transmit buffer

\ hC4 - breaklength (long), if this long is not zero the driver will transmit a break breaklength cycles,

\ the minmum lenght is 16 cycles, at 80 Mhz this is 200 nanoSeconds

\ hC8 - flags (long)

\ h_0000_0001 - if this bit is 0, CR is transmitted as CR LF

\ - if this bit is 1, CR is transmitted as CR

\

\

9.1.132 _sp

\ _sp (n1 --) put n1 in the dictionary, followed by the string in the pad

9.1.133 _udf

\ _udf (--) print out UNDEFINED WORD

9.1.134 _wc1

\ _wc1 (x -- nfa) skip blanks parse the next word and create a constant, allocate a word, 2 bytes

9.1.135 _wkeyto

\ This word variable defines the number of loops for an input timeout

9.1.136 _xasm1>1

\ _xasm1>1 (n -- n) \ the assembler operation is specified by the literal which follows (replaces the i field)

9.1.137 _xasm2>0

\ _xasm2>0 (n1 n2 --) \ the assembler operation is specified by the literal which follows (replaces the i field)

9.1.138 _xasm2>1

\ _xasm2>1 (n1 n2 -- n) \ the assembler operation is specified by the literal which follows (replaces the i field)

9.1.139 _xasm2>1IMM

\ _xasm2>1IMM (n1 n2 -- n) \ there is first an immediate word, then assembler operation

\ is specified by the literal which follows (replaces the i field)

9.1.140 _xasm2>flag

\ _xasm2>flag (n1 n2 -- n) \ the assembler operation is specified by the literal which follows (replaces the i field)

9.1.141 _xasm2>flagIMM

\ _xasm2>flagIMM (n1 n2 -- n) \ there is first an immediate word, then assembler operation

\ is specified by the literal which follows (replaces the i field)

9.1.142 _xis

\ _xis (c-addr len base -- t/f) true if the string is numeric

9.1.143 _xnu

\ _xnu (c-addr len base -- n1) convert string to a signed number

9.1.144 accept

\ accept (--) uses the pad and accepts up to padsize - 2

9.1.145 alignl

\ alignl (n1 -- n1) aligns n1 to a long (32 bit) boundary

9.1.146 alignw

\ alignw (n1 -- n1) aligns n1 to a halfword (16 bit) boundary

9.1.147 allot

\ allot (n1 --) add n1 to here, allocates space on the data dictionary or release it

9.1.148 and

\ and (n1 n2 -- n1) \ bitwise n1 and n2

9.1.149 andn

\ andn (n1 n2 -- n1) \ bitwise n1 and inverted n2

9.1.150 andnC!

\ andnC! (c1 addr --) and inverse of c1 with the contents of address

9.1.151 asmlabel

\ asmlabel (x --) skip blanks parse the next word and create an assembler entry

9.1.152 base

\ base (-- addr) access as a word, the address of the base variable

9.1.153 begin

\

9.1.154 between

\ between (n1 n2 n3 -- t/f) true if n2 <= n1 <= n3

9.1.155 bl

\ This is space constant

9.1.156 bounds

\ bounds (x n -- x+n x)

9.1.157 branch

\ branch \ 16 bit branch offset follows - -2 is to itself, +2 is next word

9.1.158 build_BootKernel

\

9.1.159 build_BootOpt

\

9.1.160 c"

\ c" (-- c-addr) compiles the string delimited by ", runtime return the addr of the counted string ** valid only in that line

\ comiple time, caddress is not left on the stack

9.1.161 c,

\ c, (x --) allocate 1 byte in the dictionary and copy x to that location

9.1.162 cappend

\ cappend (c-addr1 c-addr2 --) addpend the cstr from c-addr1 to c-addr2

9.1.163 cappendn

\ cappendn (n cstr --) print the number n and append to cstr

9.1.164 ccopy

\ ccopy (c-addr1 c-addr2 --) Copy the cstr from c-addr1 to c-addr2

9.1.165 ccreate

\ ccreate (cstr --) create a dictionary entry

9.1.166 cds

\ cds (-- addr) access as a word, the display string for this cog

9.1.167 checkdict

\ checkdict (n --) make sure there are at least n bytes available in the dictionary

9.1.168 clearkeys

\ clearkeys (--) clear the input keys

9.1.169 clkfreq

\ clkfreq (-- u1) the system clock frequency

9.1.170 cmove

\ cmove (c-addr1 c-addr2 u --) If u is greater than zero, copy u consecutive characters from the data space starting

\ at c-addr1 to that starting at c-addr2, proceeding character-by-character from lower addresses to higher addresses.

9.1.171 cnt

\ cnt - address of the the global cnt register for this cog

9.1.172 cogcds

\ cogcds (n1 -- addr) the address of the display string for cog n1

9.1.173 coghere

\ coghere (-- addr) access as a word, the first unused register address in this cog

9.1.174 cogid

\ cogid (-- n1) return id of the current cog (0 - 7)

9.1.175 cogio

\ cogio (n -- addr) the address of the data area for cog n

9.1.176 cogiochan

\ cogiochan (n1 n2 -- addr) cog n1, channel n2 ->addr

9.1.177 cognchan

\ cognchan (n1 -- n2) number of io channels for cog n2

9.1.178 cognumpad

\ cognumpad (n1 -- addr) the address of numpad for cog n1

9.1.179 cogpad

\ cogpad (n1 -- addr) the address of pad for cog n1

9.1.180 cogreset

\ cogreset (n1 --) reset the forth cog

9.1.181 cogstate

\ cogstate (n1 -- addr) the address of state for cog n1

9.1.182 cogstop

\ cogstop (n --) stop cog n

9.1.183 cogx

\ cogx (cstr n --) execute cstr on cog n

9.1.184 compile?

\ compile? (-- t/f) true if we are in a compile

9.1.185 cq

\ cq (-- addr) returns the address of the counted string following this word and increments the IP past it

9.1.186 cr

\ cr (--) emits a carriage return

9.1.187 create

\ create (--) skip blanks parse the next word and create a dictionary entry

9.1.188 cstr=

\ cstr= (cstr1 cstr2 -- t/f) case sensitive compare

9.1.189 delms

\ delms (n1 --) delay n1 milli-seconds for 80Mhz h68DB max

9.1.190 dictend

\ dictend - access as a word, the end of the total dictionary space

9.1.191 dira

\ dira - address of the the dira register for this cog

9.1.192 do

\

9.1.193 doconl

\ doconl (-- n1) \ push a 32 bit constant which follows the stack - implicit a_exit

9.1.194 doconw

\ doconw (-- h1) \ push 16 bit constant which follows on the stack - implicit a_exit

9.1.195 doloop

\

9.1.196 dothen

\

9.1.197 dovarl

\ dovarl (-- addr) \ push address of 32 bit variable which follows the stack - implicit a_exit

9.1.198 dovarw

\ dovarw (-- addr) \ push address of 16 bit variable which follows on the stack - implicit a_exit

9.1.199 dq

\ dq (--) emit a counted string at the ip, and increment the ip past it and word alignw it

9.1.200 drop

\ drop (n1 --) \ drop the value on the top of the stack

9.1.201 dup

\ dup (n1 -- n1 n1)

9.1.202 else

\

9.1.203 emit

\ emit (c1 --) emit the char on the stack

9.1.204 exec

\ exec (--) marks last entry as an eXecute word, executes always

9.1.205 execute

\ execute (addr --) execute the word - pfa address is on the stack

9.1.206 execword

\ execword (-- addr) a long, an area where the current word for execute is stored

9.1.207 exit

\ exit the current forth word, and back to the caller

9.1.208 femit?

\ femit? (c1 -- t/f) true if the output emitted a char, a fast non blocking emit

9.1.209 fill

\ fill (c-addr u char --) fill the memory with char

9.1.210 find

\ find (c-addr -- c-addr 0 | xt 2 | xt 1 | xt -1) c-addr is a counted string, 0 - not found, 2 eXecute word,

\ 1 immediate word, -1 word NOT ANSI

9.1.211 fisnumber

\ fisnumber (--) dummy routines for indirection when float package is loaded

9.1.212 fkey?

\ fkey? (-- c1 t/f) fast nonblocking key routine, true if c1 is a valid key

9.1.213 fl

\ fl (--) buffer the input and route to a free cog

9.1.214 fl_in

\ fl_in - pointer to next character for input

9.1.215 fl_lock

\ to ensure one fast load at a time

9.1.216 fnumber

\ fnumber (c-addr len -- n1) convert string to a signed number

\ dummy routines for indirection when float package is loaded

9.1.217 forthentry

\ forthentry (--) marks last entry as a forth word

9.1.218 freedict

\ freedict (--) free the forth dictionary

9.1.219 fstart

\ this word is what the IP is set to on a reboot or a reset

\ fstart (--) the start word

9.1.220 here

\ here - access as a word, the current end of the dictionary space being used

9.1.221 herelal

\ herelal (--) alignw contents of here to a long boundary, 4 byte boundary

9.1.222 herewal

\ herewal (--) align contents of here to a word boundary, 2 byte boundary

9.1.223 hex

\ hex (--) set the base for hexadecimal

9.1.224 hubopf

\ hubopf (n1 n2 -- t/f) n2 specifies which hubop (0 - 7), t/f is the 'c' flag is set from the hubop

9.1.225 hubopr

\ hubopr (n1 n2 -- n3) n2 specifies which hubop (0 - 7), n1 is the source datcog, n3 is returned,

9.1.226 i

\ i (-- n1) the most current loop counter

9.1.227 if

\

9.1.228 immediate

\ immediate (--) marks last entry as an immediate word

9.1.229 ina

\ ina - address of the the ina register for this cog

9.1.230 init_coghere

\ init_coghere (--) This word can be replaced to the assembler optimizations, initializes the coghere wvariable

9.1.231 initcon

\ initcon (--) initialize the default serial console on this cog

9.1.232 interpret

\ interpret (--) the main interpreter loop

9.1.233 interpretpad

\ interpretpad (--) interpret the contents of the pad

9.1.234 io

\ io (-- addr) the address of the io channel for the cog

9.1.235 ioconn

\ ioconn (n1 n2 --) connect the 2 cogs, disconnect them from other cogs first

9.1.236 iodis

\ iodis (n1 --) cogid to disconnect, disconnect this cog and the cog it is connected to

9.1.237 iolink

\ iolink (n1 n2 --) links the 2 cogs, output of n1 -> input of n2, output of n2 -> old output of n1

9.1.238 iounlink

\ iounlink (n1 --) unlinks the cog n1

9.1.239 isdigit

\ isdigit (c1 -- t/f) true if is it a valid digit according to base

9.1.240 isnamechar

\ isnamechar (c1 -- t/f) true if c1 is a valif name char > $20 < $7F

9.1.241 isnumber

\ isnumber (c-addr len -- t/f) true if the string is numeric

9.1.242 isunumber

\ isunumber (c-addr len -- t/f) true if the string is numeric

9.1.243 key

\ key (-- c1) get a key

9.1.244 l,

\ l, (x --) allocate 1 long, 4 bytes in the dictionary and copy x to that location

9.1.245 l>w

\ l>w (n1n2 -- n1 n2) break into 16 bits

9.1.246 lasterr

\ lasterr (-- addr) access as a char, an errorcode, set by ERR, and the kernel - if 0 - no error

9.1.247 lastnfa

\ lastnfa (-- addr) gets the last NFA

9.1.248 leave

\ leave (--) exits at the next loop or +loop, i is placed to the max loop value

9.1.249 litl

\ litl (-- n1) \ push a 32 bit literal on the stack

9.1.250 litw

\ litw (-- h1) \ push a 16 bit literal on the stack

9.1.251 lock

\ lock (lock# --)

9.1.252 lockdict

\ lockdict (--) lock the forth dictionary

9.1.253 loop

\

9.1.254 lshift

\ lshift (n1 n2 -- n3) \ n3 = n1 shifted left n2 bits

9.1.255 lxasm

\ lxasm (addr --) load the assembler at addr and execute it

9.1.256 max

\ max (n1 n2 -- n1) \ signed max of top 2 stack values

9.1.257 memend

\ memend - access as a word, the end of memory available to PropForth

9.1.258 min

\ min (n1 n2 -- n1) \ signed min of top 2 stack values

9.1.259 name=

\ name= (cstr1 cstr2 -- t/f) case sensitive compare

9.1.260 namecopy

\ namecopy (c-addr1 c-addr2 --) Copy the name from c-addr1 to c-addr2

9.1.261 namelen

\ namelen (c-addr -- c-addr+1 len) returns c-addr+1 and the length of the name at c-addr

9.1.262 namemax

\ the maximum name length allowed must be 1F

9.1.263 negate

\ negate (n1 -- 0-n1) the negative of n1

9.1.264 nextword

\ nextword (--) increment >in past current counted string

9.1.265 nfa>lfa

\ nfa>lfa (addr -- addr) go from the nfa (name field address) to the lfa (link field address)

9.1.266 nfa>next

\ nfa>next (addr -- addr) go from the current nfa to the prev nfa in the dictionary

9.1.267 nfa>pfa

\ nfa>pfa (addr -- addr) go from the nfa (name field address) to the pfa (parameter field address)

9.1.268 nfcog

\ nfcog (-- n) returns the next valid free forth cog

9.1.269 nip

\ nip (x1 x2 -- x2) delete the item x1 from the stack

9.1.270 npfx

\ npfx (c-addr1 c-addr2 -- t/f) -1 if c-addr2 is prefix of c-addr1, 0 otherwise

9.1.271 number

\ number (c-addr len -- n1) convert string to a signed number

9.1.272 numpad

\ numpad (-- addr) the of the area used by the numeric output routines, can be used carefully by other code

9.1.273 numpadsize

\ the size of the numpad, 34 bytes the largest number we can deal with is 33 digits

9.1.274 onboot

\ onboot (n1 -- n1) n1 - reset error code

9.1.275 onreset

\ onreset (n1 --) n1 - reset error code

9.1.276 or

\ or (n1 n2 -- n1_or_n2) \ bitwise or

9.1.277 orC!

\ orC! (c1 addr --) or c1 with the contents of address

9.1.278 orlnfa

\ orlnfa (c1 --) ors c1 with the nfa length of the last name field entered

9.1.279 outa

\ outa - address of the the outa register for this cog

9.1.280 over

\ over (n1 n2 -- n1 n2 n1) \ duplicate 2 value down on the stack to the top of the stack

9.1.281 pad

\ pad (-- addr) access as bytes, or words and long, the address of the pad area - used by accept for keyboard input,

\ can be used carefully by other code

9.1.282 pad>in

\ pad>in (-- addr) addr is the address to the start of the parse area.

9.1.283 pad>out

\ pad>out (-- addr) addr is the address to the the current output byte

9.1.284 padbl

\ padbl (--) fills this cogs pad with blanks

9.1.285 padsize

\ the size of the pad area, 128 bytes

9.1.286 par

\ This is the par register, always initalized to point to this cogs section of cogdata

9.1.287 parse

\ parse (c1 -- +n2) parse the word delimited by c1, or the end of buffer is reached, n2 is the length >in is the offset

\ in the pad of the start of the parsed word REPLACED IN VERSION 4.3 2011FEB24

9.1.288 parsebl

\ parsebl (-- t/f) parse the next word in the pad delimited by blank, true if there is a word

9.1.289 parsenw

\ parsenw (-- cstr) parse and move to the next word, str ptr is zero if there is no next word

9.1.290 parseword

\ parseword (c1 -- +n2) skip blanks, and parse the following word delimited by c1, update to be a counted string in

\ the pad

9.1.291 pfa>nfa

\ pfa>nfa (addr -- addr) gets the name field address (nfa) for a parameter field address (pfa)

9.1.292 prop

\ prop - access as a word, the address of the string identifier of this prop

9.1.293 propid

\ propid - access as a word, the numeric id of this prop

9.1.294 r>

\ r> (-- n1) \ pop top of RS to stack

9.1.295 reboot

\ reboot (--) reboot the propellor chip

9.1.296 reset

\ reset (--) reset this cog

9.1.297 rot

: rot h2 ST@ h2 ST@ h2 ST@ 3 ST! 3 ST! 0 ST! ;

9.1.298 rot2

\ rot2 (x1 x2 x3 -- x3 x1 x2)

9.1.299 rshift

\ rshift (n1 n2 -- n3) \ n3 = n1 shifted right logically n2 bits

9.1.300 serial

\ serial (n1 n2 n3 --)

\ n1 - tx pin

\ n2 - rx pin

\ n3 - baud rate

\

\ h00 - h04 -- io channel

\ h04 - h84 -- the receive buffer

\ h84 - hC4 -- the transmit buffer

\ hC4 - breaklength (long), if this long is not zero the driver will transmit a break breaklength cycles,

\ the minmum lenght is 16 cycles, at 80 Mhz this is 200 nanoSeconds

\ hC8 - flags (long)

\ h_0000_0001 - if this bit is 0, CR is transmitted as CR LF

\ - if this bit is 1, CR is transmitted as CR

\

9.1.301 seti

\ seti (n1 --) set the most current loop counter

9.1.302 skipbl

\ skipbl (--) increment >in past blanks or until it equals padsize

9.1.303 space

\ space (--) emits a space

9.1.304 spaces

\ spaces (n --) emit n spaces

9.1.305 state

\ state (-- addr) access as a char

\ bit 0 - 0 - interpret mode / 1 - forth compile mode

\ bit 1 - 0 - prompts and errors on / 1 - prompts and errors off

\ bit 2 - 0 - Other / 1 - PropForth cog

\ bit 3 - 0 - accept echos chars on / 1 - accept echos chars off

\ bit 4 - 0 - accept echos line off / 1 - accept echos line on

\ bit 5 - 7 - number of io channels - 1

9.1.306 swap

\ swap (n1 n2 -- n2 n1) \ swap top 2 stack values

9.1.307 t0

\ these are temporay variables, and by convention are only used within a word

\ caution, make sure you know what words you are calling

\ t0 - access as a word, temp variable

9.1.308 t1

\ these are temporay variables, and by convention are only used within a word

\ caution, make sure you know what words you are calling

\ t1 - access as a word, temp variable

9.1.309 tbuf

\ these are temporay variables, and by convention are only used within a word

\ caution, make sure you know what words you are calling

\ tbuf - access as a chars, words, or longs. Temp array of 32 bytes

9.1.310 then

****

9.1.311 thens

****

9.1.312 tochar

\ tochar (n1 -- c1) convert c1 to a char

9.1.313 todigit

\ todigit (c1 -- n1) converts character to a number

9.1.314 tuck

\ tuck (x1 x2 -- x2 x1 x2)

9.1.315 u*

\ u* (u1 u2 -- u1*u2) u1*u2 must be a valid 32 bit unsigned number

9.1.316 u.

\ u. (n1 --) prints the unsigned number on the top of the stack

9.1.317 u/

\ u/ (u1 u2 -- u1/u2) u1 divided by u2

9.1.318 u/mod

\ u/mod (u1 u2 -- remainder quotient) both remainder and quotient are 32 bit unsigned numbers

9.1.319 u>=

\ u>= (u1 u2 -- t/f) \ flag is true if and only if u1 is greater or equal to than u2

9.1.320 um*

\ um* (u1 u2 -- u1*u2L u1*u2H) \ unsigned 32bit * 32bit -- 64bit result

9.1.321 um/mod

\ um/mod (u1lo u1hi u2 -- remainder quotient) \ unsigned divide & mod u1 divided by u2

9.1.322 unlock

\ unlock (lock# --)

9.1.323 unlockall

\ unlockall (--) unlocks everything this cog has locked

9.1.324 until

\

9.1.325 unumber

\ unumber (c-addr len -- u1) convert string to an unsigned number

9.1.326 version

\ version - access as a word, the address of the string version of PropForth

9.1.327 w,

\ w, (x --) allocate 1 halfword 2 bytes in the dictionary and copy x to that location

9.1.328 w>l

\ w>l (n1 n2 -- n1n2) consider only lower 16 bits of each source word

9.1.329 wconstant

\ wconstant (x --) skip blanks parse the next word and create a constant, allocate a word, 2 bytes

9.1.330 wlastnfa

\ wlastnfa - access as a word, the address of the last nfa

9.1.331 wvariable

\ wvariable (--) skip blanks parse the next word and create a variable, allocate a word, 2 bytes

9.1.332 xisnumber

\ xisnumber (c-addr len -- t/f) true if the string is numeric

9.1.333 xnumber

\ xnumber (c-addr len -- n1) convert string to a signed number

9.1.334 xor

\ \ xor (n1 n2 -- n1_xor_n2) \ bitwise xor

9.1.335 {

\ { (--) discard all the characters between { and }

\ open brace MUST be the first and only character on a new line, the close brace must be on another line

9.1.336 }

\ } (--)

9.2 DevKernel Word Set (BuildDevKernel)

****

9.2.1 #C

\ #C (c1 --) prepend the character c1 to the number currently being formatted

9.2.2 $C_a__xasm2>1

\ A word constant which is an address in the PropForth assembler kernel

9.2.3 $C_a__xasm2>1IMM

\ A word constant which is an address in the PropForth assembler kernel

9.2.4 $C_a_doconl

\ A word constant which is an address in the PropForth assembler kernel

9.2.5 $C_a_dovarl

\ A word constant which is an address in the PropForth assembler kernel

9.2.6 $C_rsPtr

\ A word constant which is an address in the PropForth assembler kernel

9.2.7 $C_rsTop

\ A word constant which is an address in the PropForth assembler kernel

9.2.8 $C_stPtr

\ A word constant which is an address in the PropForth assembler kernel

9.2.9 $C_stTOS

\ A word constant which is an address in the PropForth assembler kernel

9.2.10 $C_stTop

\ A word constant which is an address in the PropForth assembler kernel

9.2.11 (dumpb)

\ Internal word used by the dump words

9.2.12 (dumpe)

\ Internal word used by the dump words

9.2.13 (dumpm)

\ Internal word used by the dump words

9.2.14 (forget)

\ (forget) (cstr --) wind the dictionary back - caution

9.2.15 *

\ * (n1 n2 -- n1*n2) n1 multiplied by n2

9.2.16 */

\ */ (n1 n2 n3 -- n4) n4 = (n1*n2)/n3. Uses a 64bit intermediate result.

9.2.17 */mod

\ */mod (n1 n2 n3 -- n4 n5) n5 = (n1*n2)/n3, n4 is the remainder. Uses a 64bit intermediate result.

9.2.18 .byte

\ .byte (n1 --) output a byte

9.2.19 .cogch

\ .cogch (n1 n2 --) print as x(y)

9.2.20 .con

\ .con (n1 --) print n1 to the console, non blocking, may get overwritten

9.2.21 .conbyte

\ .con (c1 --) print byte c1 to the console, non blocking, may get overwritten

9.2.22 .concr

\ .concr (--) emit a cr to the console, non blocking, may get overwritten

9.2.23 .concstr

\ .concstr (cstr --) emit cstr to console, non blocking, may get overwritten

9.2.24 .conemit

\ .conemit (c1 --) emit cr to console, non blocking, may get overwritten

9.2.25 .conlong

\ .conlong (n1 --) print n1 to console, non blocking, may get overwritten

9.2.26 .const?

\ .const? (--) prints out the stack to the console, non blocking, may get overwritten

9.2.27 .conwait

\ .conwait (--) if con is not ready for a char, wait long enough for a char to transmit

9.2.28 .conword

\ .conword (n1 --) print n1 to console, non blocking, may get overwritten

9.2.29 .long

\ .long (n1 --) output a long

9.2.30 .word

\ .word (n1 --) output a word

9.2.31 /

\ / (n1 n2 -- n1/n2) n1 divided by n2

9.2.32 /mod

\ /mod (n1 n2 -- n3 n4) \ signed divide & mod n4 = n1/n2, n3 is the remainder

9.2.33 1lock

\ 1lock(--) equivalent to 1 lock

9.2.34 1unlock

\ 1unlock(--) equivalent to 1 unlock

9.2.35 2*

\ 2* (n1 -- n1<<1) n2 is shifted logically left 1 bit

9.2.36 4-

\ 4- (n1 -- n1-4)

9.2.37 4/

\ 4/ (n1 -- n1>>2) n2 is shifted arithmetically right2 bits

9.2.38 EC@

\ EC@ (eeAddr -- c1) read a byte from the eeprom

9.2.39 EW!

\ EW! (n1 eeAddr --) write n1 to the eeprom

9.2.40 EW@

\ EW@ (eeAddr -- n1) read a word from the eeprom

9.2.41 _bf

\ _bf (n1 -- cstr) format n1 as a byte

9.2.42 _eestart

\ _eestart (--) start the data transfer

9.2.43 _eestop

\ _eestop (--) stop the data transfer

9.2.44 _ft

\ _ft (n1 divisor -- cstr) internal format routine

9.2.45 _lf

\ _lf (n1 -- cstr) format n1 as a long

9.2.46 _nd

\ _nd (n1 -- n2) internal format routine

9.2.47 _pna

\ _pna (pfa --) print the address, contents and forth name

9.2.48 _sclh

\ _sclh (--) eeprom clk hi

9.2.49 _scli

\ _scli (--) eeprom clk in

9.2.50 _scll

\ _scll (--) eeprom clk lo

9.2.51 _sclo

\ _scll (--) eeprom clk out

9.2.52 _sda?

\ _sda? (-- t/f) read the state of the sda pin

9.2.53 _sdah

\ _sdah (--) eeprom sda hi

9.2.54 _sdai

\ _sdai (--) eeprom sda in

9.2.55 _sdal

\ _sdal (--) eeprom sda lo

9.2.56 _sdao

\ _sdao (--) eeprom sda out

9.2.57 _wf

\ _wf (n1 -- cstr) format n1 as a word

9.2.58 _words

\ _words (cstr --) prints the words in the forth dictionary starting with cstr, 0 prints all

9.2.59 abs

\ abs (n1 -- abs_n1) absolute value of n1

9.2.60 andC!

\ andC! (c1 addr --) and c1 with the contents of address

9.2.61 build?

\ build? (--) print out build information

9.2.62 build_DevKernel

\ Word constant

9.2.63 cog?

\ cog? (--) print out cog information

9.2.64 cogdump

\ cogdump (adr cnt --) dump cog memory

9.2.65 constant

\ constant (x --) skip blanks parse the next word and create a constant, allocate a long, 4 bytes

9.2.66 decimal

\ decimal (--) set the base for decimal

9.2.67 dump

\ dump (adr cnt --) dump main memory, uses tbuf

9.2.68 edump

\ edump (adr cnt --) dump eeprom, uses tbuf

9.2.69 eereadpage

\ the eereadpage and eewritePage words assume the eeprom are 64kx8 and will address up to

\ 8 sequential eeproms

\ eereadpage (eeAddr addr u -- t/f) return true if there was an error, use lock 1

9.2.70 eewritepage

\ the eereadpage and eewritePage words assume the eeprom are 64kx8 and will address up to

\ 8 sequential eeproms

\ eewritepage (eeAddr addr u -- t/f) return true if there was an error, use lock 1

9.2.71 forget

\ forget (--) wind the dictionary back to the word which follows - caution

9.2.72 free

\ free (--) display free main bytes and current cog longs

9.2.73 ibound

\ ibound (-- n1) the upper bound of i

9.2.74 invert

\ invert (n1 -- n2) bitwise invert n1

9.2.75 io>cogchan

\ io>cogchan (addr -- n1 n2) addr -> n1 cogid, n2 channel

9.2.76 j

\ j (-- n1) the second most current loop counter

9.2.77 lasti?

\ lasti? (-- t/f) true if this is the last value of i in this loop, assume an increment of 1

9.2.78 lock?

\ lock? (--) displays the status of the locks

9.2.79 onreset

\ onreset (n1 --) reset message and error n1 on a reset, echo to the console

9.2.80 pfa?

\ pfa? (addr -- t/f) true if addr is a pfa

9.2.81 pinhi

\ pinhi (n1 --) set pin # n1 to hi

9.2.82 pinin

\ pinin (n1 --) set pin # n1 to an input

9.2.83 pinlo

\ pinlo (n1 --) set pin # n1 to lo

9.2.84 pinout

\ pinout (n1 --) set pin # n1 to an output

9.2.85 px

\ px (t/f n1 --) set pin # n1 to h - true or l false

9.2.86 px?

\ px? (n1 -- t/f) true if pin n1 is hi

9.2.87 rev

\ rev (n1 n2 -- n3) n3 is n1 with the lower 32-n2 bits reversed and the upper bite cleared

9.2.88 revb

\ revb (n1 -- n2) n2 is the lower 8 bits of n1 reveresed

9.2.89 rnd

\ rnd (-- n1) n1 is a random number from 00 - FF

9.2.90 rndtf

\ rndtf (-- t/f) true or false randomly

9.2.91 rs?

\ rs? (--) prints out the return stack

9.2.92 saveforth

\ saveforth(--) write the running image to eeprom UPDATES THE CURRENT VERSION STR

9.2.93 sc

\ sc (--) clears the stack

9.2.94 serflags?

\ serflags? (n1 -- n2) n2 are the serial flags for the serial driver running on cog n1

9.2.95 sersendbreak

\ sersendbreak (n2 n1 --) for the serial driver running on cog n1, send a break of n2 clock cycles

9.2.96 sersetflags

\ sersetflags (n2 n1 -- O) for the serial driver running on cog n1, set the flags to n2

9.2.97 sign

\ sign (n1 n2 -- n3) n3 is the xor of the sign bits of n1 and n2

9.2.98 st?

\ st? (--) prints out the stack

9.2.99 u*/

\ u*/ (u1 u2 u3 -- u4) u4 = (u1*u2)/u3 Uses a 64bit intermediate result.

9.2.100 u*/mod

\ u*/mod (u1 u2 u3 -- u4 u5) u5 = (u1*u2)/u3, u4 is the remainder. Uses a 64bit intermediate result.

9.2.101 variable

\ variable (--) skip blanks parse the next word and create a variable, allocate a long, 4 bytes

9.2.102 waitcnt

\ waitcnt (n1 n2 -- n1) \ wait until n1, add n2 to n1

9.2.103 waitpeq

\ waitpeq (n1 n2 --) \ wait until state n1 is equal to ina anded with n2

9.2.104 waitpne

\ waitpne (n1 n2 --) \ wait until state n1 is not equal to ina anded with n2

9.2.105 words

\ words (--) prints the words in the forth dictionary, if the pad has another string following, with that prefix

9.3 EEpromKernel Word Set (build_fsrd, build_fswr)

\ A very simple file system for eeprom. The goal is not a general file system, but a place to put text

\ (or code) in eeprom so it can be dynamically loaded by propforth

\

\ the eeprom area start is defined by fsbot and the top is defined by fstop

\ The files are in eeprom memory as such:

\ 2 bytes - length of the contents of the file

\ 1 byte - length of the file name (this is a normal counted string, counted strings can be up

\ 255 bytes in length, the length here is limited for space reasons)

\ 1 - 31 bytes - the file name

\ 0 - 65534 bytes - the contents of the file

\

\ the last file has a length of 65535 (0hFFFF)

\

\ the start of every file is aligned with eeprom pages, for efficient read and write, this is 64 bytes

\

\ Status: 2010NOV24 Beta

\

\ 2011FEB03 - fix to align page reads to page addresses so crossing eeproms does not cause a problem _fsread

\

\ 2011MAY31 - stable, reformat, updated error codes, added error message for file not found, added RO option

\

\ main routines

\

\ fsload filename - reads filename and directs it to the next free forth cog, every additional nested fsload

\ requires an additional free cog

\ fsread filename - reads the file and echos it directly to the terminal

\ fswrite filename - writes the file to the filesystem - takes input from the input until ...\0h0d is encounterd

\ - ie 3 dots followed by a carriage return

\ fsls - lists the files

\ fsclear - erases all files

\ fsdrop - erases the last file

9.3.1 _fnf

\ _fnf (--) file not found message

9.3.2 _fsfind

\ _fsfind (cstr -- addr) find the last file named cstr, addr is the eeprom address, 0 if not found

9.3.3 _fsfree

\ _fsfree (-- n1) n1 is the first location in the file system, -1 if there are none

9.3.4 _fsk

\ _fsk (n1 -- n2) n1<<8 or a key from the input

9.3.5 _fslast

\ _fslast (-- addr) find the last file, 0 if not found

9.3.6 _fsload

\ _fsload (cstr --) load the using the next free cog

9.3.7 _fsnext

\ _fsnext (addr1 -- addr2 t/f) addr - the current file address, addr2 - the next addr, t/f - true if we have

\ gone past the end of the eeprom. t0 -length of the current file

\ t1 - length of the file name (char)

9.3.8 _fsp

\ _fsp filename (-- cstr) filename, if cstr is 0 no file found

9.3.9 _fspa

\ _fspa (addr1 -- addr2) addr2 is the next page aligned address after addr1

9.3.10 _fsrd

\ _fsrd (addr1 addr2 n1 --) addr1 - the eepropm address to read, addr2 - the address of the read buffer

\ n1 - the number of bytes to read

9.3.11 _fsread

\ _fsread (cstr --) read file to output

9.3.12 _fswr

\ _fswr (addr1 addr2 n1 --) addr1 - the eepropm address to write, addr2 - the address to write from

\ n1 - the number of bytes to write

9.3.13 build_fsrd

\ Word constant

9.3.14 build_fswr

\ Word constant

9.3.15 fsbot

\ Long constant - the start adress in eeprom for the file system

9.3.16 fsclear

\ fsclear (--) erase all files and initialize the eeprom file system

9.3.17 fsdrop

\ fsdrop (--) deletes last file

9.3.18 fsfree

\ fsfree (--) print out free bytes in the eeprom file system

9.3.19 fsload

\ fsload filename (--) send the file to the next free forth cog

9.3.20 fsls

\ fsls (--) list the files

9.3.21 fsps

\ fsps - word constant, the page size set to 64, a page size which should work with 32kx8 & 64kx8 eeproms and should work with larger as well. MUST BE A POWER OF 2

9.3.22 fsread

\ fsread filename (--) prints filename to the output

9.3.23 fstop

\ Long constant - the end address in the eeprom for the file system

9.3.24 fswrite

\ fswrite filename (--) writes a file until ... followed immediately by a cr is encountered

9.3.25 onboot

\ onboot (n1 -- n1) execute file boot.f if it exists

9.4 SDKernel Word Set (build_sd)

sdfs is meant to be a very simple fast file system on top of sd_driver.

The design criteria are around small code size and speed, as opposed to generality and versatility.

Main concepts:

1. A file system occupies n contiguous 512 byte blocks on the SD card. If it is desired that the card should be FAT32 formatted, it should be possible to format card, create a very large file, figure out which blocks the file occupies, and mount sdfs to use those blocks.

This means sdfs could be manipulated on a pc by writing some code. This is NOT a goal of the initial implementation.

The file system must not start at block 0, block 1 is the first valid block number.

This also means multiple file systems can be defined on one sd card, useful for isolating functions like logging.

An sd card can be partitioned into multiple "disks", each which can contain one file system. There is no partition table or disk routines, when a filesystem is created, the start and end inherently defines the partion.

2. Files are 2 - n contiguous blocks, the maximum file size is 2Gig, (max positive 32 bit integer.) When a file is created, the space that is allocated to the file (the space is allocated as blocks of 512 bytes) is the maximum size the file can grow to. This trades space efficiency

for a very simple and fast allocation.

3. File names must be 1 to 26 characters in length, and can only contain characters 0h30 - 0h7D. There are no other restrictions on file names. It is recommended that names do not use special characters. The reason for this is mapping urls to file names gets more complicated.

4. There is directory support, a directory is a fixed length file, whose name ends with a / There is no other differentiator. A directory can contain up to 2048 entries. A simple hashing mechanism makes navigation quick. (This hash function should be tested more thouroughly for at some point, but initial testing seems ok.) This optimizes for opening files, the result is that to list a directory the whole directory must be traversed, so directory "listing" is slower.

However, assuming the hash function performs reasonably, finding a file, and opening the file file be fast, and not slow down as there are more entries in the directory.

The root directory is /

Pathnames are simple concatenated directory and file names.

The maximum total length of a path name is 120 characters, this should make mapping urls to files very easy, and allow the use of the pad for parsing and file name manipulation.

To access a file, or directory, the current directory must be set to the parent of the file or directory. File access is only in that directory. This means no relative navigation or fully qualified file names. Reasons are 2 fold, 1 simplicity, 2 security. This will bebome evident when the http server is using the file system.

5. There is no limit on directory depth other than that which is imposed by the maximum path name length. The deeper a file is in the structure, the longer it will take to navigate to it.

6. Each file has a header block, immediately followed by all the data blocks.

7. The header block number is used by routines to reference files and directories.

8. Block numbers are absolute, this makes debugging, repair, and verification much simpler.

Unfortunately it means you cannot easily move the file system around.

9. The main interface routines to sdfs are:

```
\ _sd_CrEaTe ( n1 n2 -- ) n1 - starting block, n2 - last block + 1, CREATE a file system, WIPES OUT DATA



\ sd_mount ( n1 -- ) mount the file system, n1 - the starting block of the file system,

\ the file system must be mounted before it is used



\ sd_createdir ( cstr -- n1 ) cstr is the name of the directory to create, n1 is the header block
```

\ of the directory. If this directory already exists, it returns the root block of the existing

\ directory

\ sd_cd.. (--) make the parent directory the current directory

\ sd_cd (cstr --) make cstr the current directory, if it does not exists, nothing happens

\ the directory name in cstr must have a / at the end of it

\ sd_cwd (--) get the pathname of the current directory and copy it to pad

\ sd_ls (--) list the current directory

\ sd_createfile (cstr n1 -- n2) allocate n1 blocks, this includes the header block

\ sd_find (filename -- blocknumber/0) search for filename in the current directory

\ sd_stat (filename --) prints stats for the file

\ sd_write (numblocks filename --) writes a new or existing file until ... followed immediately by a cr is encountered

\ if the file exists, writing will be truncated to the existing maximum file size allocated when the file was created

\ sd_trunc (length filename --) sets the number of bytes used in the file to length

\ sd_appendblk(addr size blk --)

\ sd_append(addr size filename --)

\ sd_readblk (n1 --) n1 - header block number of file,

\   read the file and emit the char

\ sd_read (filename --) read the file and emit the chars

\ sd_loadblk (n1 --) n1 - the header block for the fileloads the file,

\   routes the file to the next free forth cog

\ sd_load (cstr --) loads the file, routes the file to the next free forth cog

\

\ These are provided for command line convenience

\

\ ls (--)

\ cd dirname (--)

\ cd.. (--)

\ cd/ (--)

\ cwd (--) print the current directory

\ mkdir dirname (--)

\ fread filename (--)

\ fcreate filename (numblocks_to_allocate --)

\ fwrite filename (numblocks_to_allocate --)

\ fstat filename (--)

9.4.1 $S_sd_clk

\ Word constant, io pins connecting to the sd card

9.4.2 $S_sd_cs

\ Word constant, io pins connecting to the sd card

9.4.3 $S_sd_di

\ Word constant, io pins connecting to the sd card

9.4.4 $S_sd_do

\ Word constant, io pins connecting to the sd card

9.4.5 .num

\ .num (n1 --) print n1 as a fixed format number

9.4.6 _fnf

\ _fnf (--) file not found message

9.4.7 _fsk

\ _fsk (n1 -- n2) n1<<8 or a key from the input

9.4.8 _nf

\ _nf (n1 -- cstr) formats n1 to a fixed format 16 wide, leading spaces variable, one trailing space

9.4.9 _readlong

\ _readlong (addr -- long) reads an unaligned long

9.4.10 _sd_alloc

\ _sd_alloc (n1 -- n2) n1 - number of blocks to allocate, n2 - starting block, assumes v_currentdir is valid

9.4.11 _sd_appendbytes

\ _sd_appendbytes (src byteoffset nbytes -- updatedsrc)

\ t0 - nybtes

\ t1 - byteoffset

\ tbuf - src

9.4.12 _sd_ccs

\ \ this variable is a reflection of the ccs bit in the OCR

\ \ register, if 0, we send byte addresses

\ \ (block aligned) to the card, otherwise block addresses,

\ \ only used internally in the driver

\ \ initialized by sd_init

9.4.13 _sd_clk_out

\ basic pin functions to drive the sd card

9.4.14 _sd_clk_out_h

\ basic pin functions to drive the sd card

9.4.15 _sd_clk_out_l

\ basic pin functions to drive the sd card

9.4.16 _sd_cmdr16

\ _sd_cmdr16 (crc arg cmd -- n1) send the command, wait for response, n1 - the 16 bit response

\ n1 is set to all 1's (FFFFFFFF or -1) in the case of a timeout

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.17 _sd_cmdr40

\ _sd_cmdr40 (crc arg cmd -- n1 n2) send the command, wait for response, n1 - the 8 bit response,

\ n2 - 32 bit response, n1 and n2 are set to all 1's (FFFFFFFF or -1) in the case of a timeout

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.18 _sd_cmdr8

\ _sd_cmdr8 (crc arg cmd -- n1) send the command, wait for response, n1 - the 8 bit response

\ n1 is set to all 1's (FFFFFFFF or -1) in the case of a timeout

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.19 _sd_cmdr8data

\ _sd_cmdr8data (datalen crc arg cmd -- n1) send the command, wait for response, the read the data into _sd_buf,

\ n1 - the 8 bit response n1 is set to all 1's (FFFFFFFF or -1) in the case of a timeout

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.20 _sd_cogend

\ set the data area for the buffer at the end of the assembler code, the allocation is done in sd_init

\ _sd_cogend is the end of the buffer

9.4.21 _sd_cs_out

\ basic pin functions to drive the sd card

9.4.22 _sd_cs_out_h

\ basic pin functions to drive the sd card

9.4.23 _sd_cs_out_l

\ basic pin functions to drive the sd card

9.4.24 _sd_di_out

\ basic pin functions to drive the sd card

9.4.25 _sd_di_out_h

\ basic pin functions to drive the sd card

9.4.26 _sd_di_out_l

\ basic pin functions to drive the sd card

9.4.27 _sd_dn

****

9.4.28 _sd_do_in

\ basic pin functions to drive the sd card

9.4.29 _sd_fsp

****

9.4.30 _sd_hash

\

9.4.31 _sd_hc

\ \ this variable is a reflection of the CSD structure

\ \ version bits in the CSD register,

\ \ if 0, it is a Standard card, otherwise a High or

\ \ Extended Capacity card

\ \ initialized by sd_init

9.4.32 _sd_init

\ _sd_init (--) initialize the SD card, this routine is only called once for the propeller

9.4.33 _sd_initdir

\ _sd_initdir (n1 --) n1, directory block number, initialize the directory

9.4.34 _sd_initialized

\ Word variable, 0 if the SD card is not initialized

9.4.35 _sd_maxblock

\ \ this long variable is the maximum block number for the card

\ \ initialized by sd_init (card capacity is this * 512)

9.4.36 _sd_readdata

\ _sd_readdata(n1 --) n1 number of bytes to read, n1 must be a multilpe of 4, and it does not include the crc,

\ the crc is discarded by this routine. The data is read to sd_cogbuf

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.37 _sd_setdirentry

\ _sd_setdirentry (filename blocknumber --) write the directory entry

9.4.38 _sd_shift_in

\ _sd_shift_in (-- c1) shift in one char, most siginificant bit first

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.39 _sd_shift_inlong

\ _sd_shift_inlong (-- n1) shift in one long, most siginificant bit first

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.40 _sd_shift_out

\ _sd_shift_out (c1 --) shift out one char most siginficant bit first

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.41 _sd_shift_outlong

\ _sd_shift_outlong (n1 --) shift out one long most siginficant bit first

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.42 _sd_writedata

\ _sd_writedata(n1 --) n1 number of bytes to write, n1 must be a multilpe of 4, and it does not include the crc,

\ the crc is written as FF by this routine. The data is written from sd_cogbuf

There are 2 versions of this word, the first is a slow version used during initialization. The second is a fast assembler version used after the SD card is initialized.

9.4.43 a_shift

\ Internal word, used to invoke the assembler routines

9.4.44 build_sd

\ Word constant

9.4.45 cd

\ cd dirname (--)

9.4.46 cd..

\ cd.. (--)

9.4.47 cd/

\ cd/ (--)

9.4.48 cog>mem

\ cog>mem (memaddr cogaddr numlongs --) memaddr must be long aligned

9.4.49 cog>pad

\ cog>pad (n1 --) the cog address to start reading 32 longs

9.4.50 cog>tbuf7

\ tbuf>cog7 (n1 --) the cog address to start writing 7 longs

9.4.51 cwd

\ cwd (--) print the current directory

9.4.52 fcreate

\ fcreate filename (numblocks_to_allocate --)

9.4.53 fload

\ fload filename (--) load the file using the next free cog

9.4.54 fread

\ fread filename (--) print the file to the ouput

9.4.55 fstat

\ fstat filename (--) print the file stats to the output

9.4.56 fwrite

\ fwrite filename (numblocks_to_allocate --) allocate blocks and write until a ... followed by a cr is encountered

9.4.57 ls

\ ls (--) list the files

9.4.58 mem>cog

\ mem>cog (memaddr cogaddr numlongs --) memaddr must be long aligned

9.4.59 mkdir

\ mkdir dirname (--) create the directory

9.4.60 onboot

\ onboot (n1 -- n1) load the file sd_boot.f

9.4.61 pad>cog

\ pad>cog (n1 --) the cog address to start writing 32 longs

9.4.62 sd_append

\ sd_append(addr size filename --) append buffer of size to the file

9.4.63 sd_appendblk

\ sd_appendblk(addr size headerblk --) append buffer of size to the file at headerblk

9.4.64 sd_blockread

\ sd_blockread (n1 --) n1 - the block number. Reads a 512 byte block into _sd_buf

9.4.65 sd_blockwrite

\ sd_blockwrite (n1 --) n1 - the block number. Writes 512 byte block from _sd_buf

9.4.66 sd_cd

\ sd_cd (cstr --) make cstr the current directory, if it does not exists, nothing happens

\ the directory name in cstr must have a / at the end of it

9.4.67 sd_cd..

\ sd_cd.. (--) make the parent directory the current directory

9.4.68 sd_cogbuf

\ set the data area for the buffer at the end of the assembler code, the allocation is done in sd_init

\ sd_cogbuf is the beginning of the buffer

9.4.69 sd_cogbufclr

\ sd_cogbufclr (--) initialize the buffer to zeros

9.4.70 sd_createdir

\ sd_createdir (cstr -- n1) cstr is the name of the directory to create, n1 is the header block

\ of the directory. If this directory already exists, it returns the root block of the existing

\ directory

9.4.71 sd_createfile

\ sd_createfile (cstr n1 -- n2) allocate n1 blocks, this includes the header block,

\ create a directory entry, and write the file header,

\ n2 is the block number of the file header

9.4.72 sd_cwd

\ sd_cwd (--) get the pathname of the current directory and copy it to pad

9.4.73 sd_find

\ sd_find (filename -- blocknumber/0) search for filename in the current directory

9.4.74 sd_init

\ sd_init (--) call for every cog using the sd card

9.4.75 sd_load

\ sd_load (cstr --) loads the file, routes the file to the next free forth cog

9.4.76 sd_loadblk

\ sd_loadblk (n1 --) n1 - the header block for the fileloads the file,

\ routes the file to the next free forth cog

9.4.77 sd_lock

\ sd_lock (--) lock the sd card so no one else can use it

9.4.78 sd_ls

\ sd_ls (--) list the current directory

9.4.79 sd_mount

\ sd_mount (n1 --) mount the file system, n1 - the starting block of the file system,

\ the file system must be mounted before it is used

9.4.80 sd_read

\ sd_read (filename --) read the file and emit the chars

9.4.81 sd_readblk

\

\ sd_readblk (n1 --) n1 - header block number of file,

\ read the file and emit the chars

9.4.82 sd_stat

\ sd_stat (filename --) prints stats for the file

9.4.83 sd_trunc

\ sd_trunc (length filename --) sets the number of bytes used in the file to length

9.4.84 sd_uninit

\ sd_uninit (--) "releases" the cog memory

9.4.85 sd_unlock

\ sd_unlock (--) unlock the sd card

9.4.86 sd_write

\ sd_write (numblocks filename --) writes a new or existing file until ... followed immediately by a cr is encountered

\ if the file exists, writing will be truncated to the existing maximum file size allocated when the file was created

9.4.87 tbuf>cog7

\ tbuf>cog7 (n1 --) the cog address to start writing 7 longs

9.4.88 v_currentdir

\ Address to a long in cog memory, the block number of the current directory

9.4.89 v_sd_clk

\ Address to a long in cog memory, a mask with the for sd_clk

9.4.90 v_sd_di

\ Address to a long in cog memory, a mask with the for sd_di

9.4.91 v_sd_do

\ Address to a long in cog memory, a mask with the for sd_do

9.4.92 v_sdbase

\ Address to a long in cog memory, the base for all the cog memory used by thesd filesystem