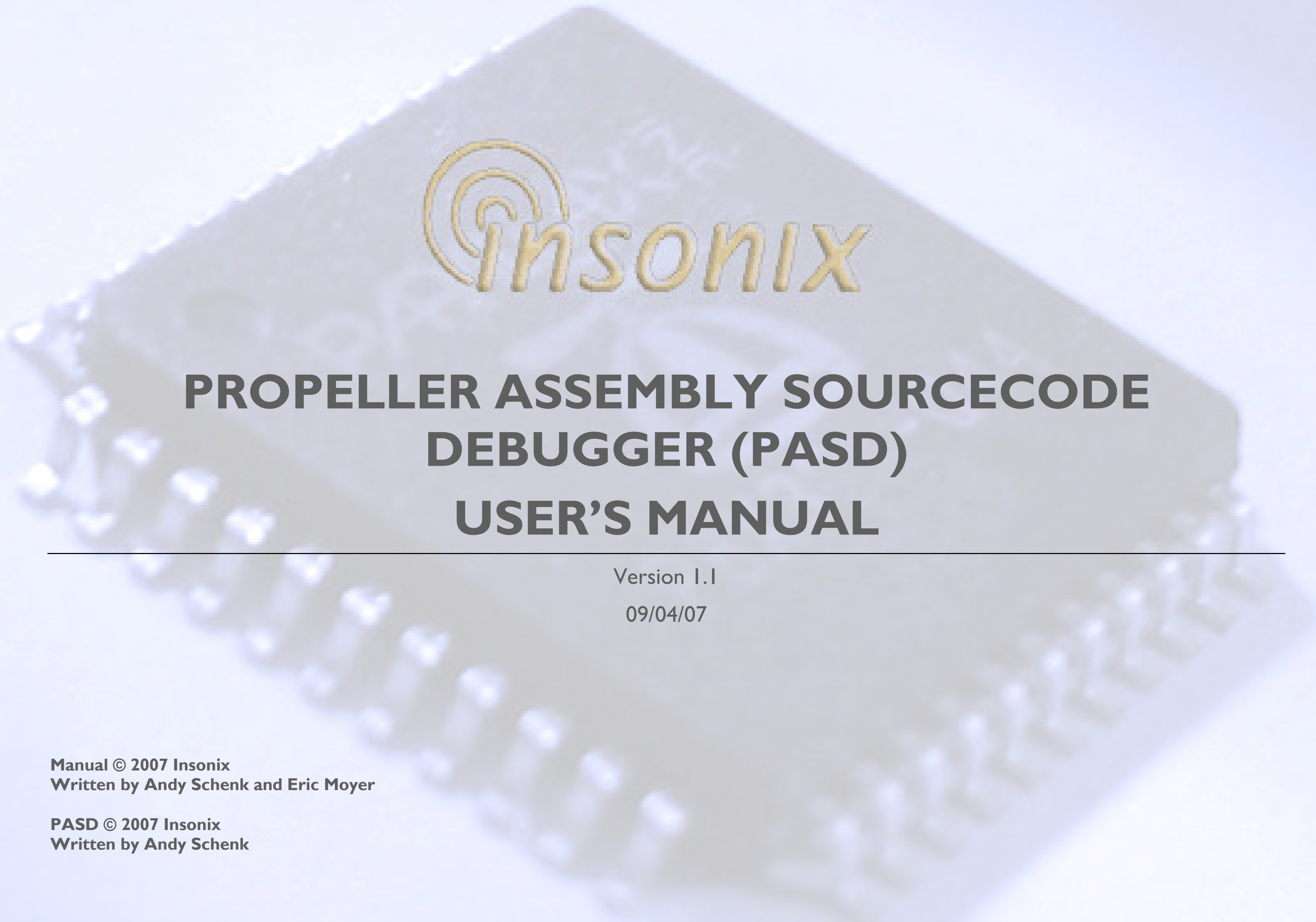


PROPELLER ASSEMBLY SOURCECODE DEBUGGER (PASD)

USER'S MANUAL

Version 1.1

09/04/07

TABLE OF CONTENTS

“There are no rules here--we're trying to accomplish something."”

-Thomas Alva Edison

1. PASD Overview

The Propeller Assembler Source-code Debugger (PASD) is a suite of software components which enable end-users to debug Propeller assembly language code at the source level using a remote (USB attached) Windows PC. PASD supports setting multiple break points, single-step execution, memory inspection/modification of COG RAM, inspection of Main RAM, label recognition, and I/O pin state inspection.

The debugger suite consists of a Windows application, a spin object and a short Debug Kernel which must be inserted at the beginning of the code to be debugged. The Debug Kernel is only 12 longs in size, and makes possible communication with the PASD spin driver, which runs into own Cog. The PASD spin driver communicates over the Propeller's serial programming interface with the PASD Windows application running on an attached PC. Except for pins 30 and 31 (the Propeller's serial programming interface pins) all Propeller IO pins are freely available during debugging.

The total Propeller resource footprint of the PASD suite is:

1) Two IO Pins (30 and 31, the serial programming interface pins).

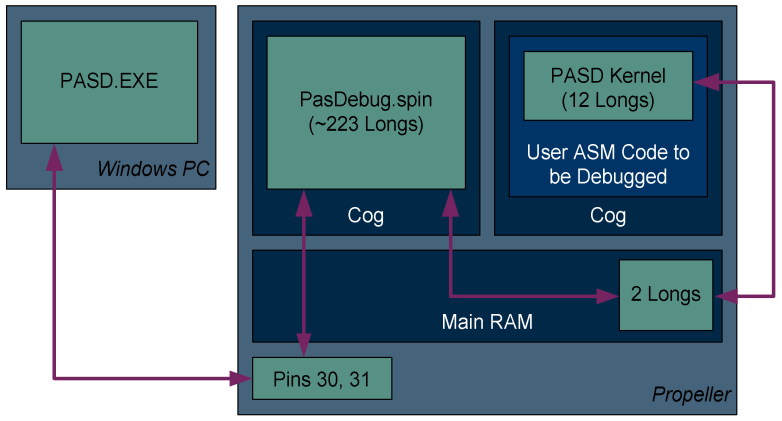
2) 12 longs at the start of COG Ram in the COG whose assembly code is being debugged.

3) The upper two longs of Main RAM ($7ff8 and $7fff).

4) The PASD driver which occupies about 223 Longs and runs in one dedicated COG.

All remaining Propeller resources (cogs, RAM, pins) are fully usable.

PASD presently supports debugging code in only one COG at a time. It would be conceivable for a future version of PASD to support debugging in all remaining cogs.



2. Installation

Place the contents of the PASD .ZIP file into any directory. PASD.exe is a stand alone executable and does not unpack any files or require a "Setup" installation.

3. PASD Walkthrough

1) Double click the **PASD_AsmDebugDemo.spin** module to load it into the Propeller IDE.

PASD_AsmDebugDemo.spin
Propeller Source Code
3 KB

2) *Note: If you are working on a Hydra or target hardware other then the Propeler Demo Board then you must modify the **_clkmode** and **_xinfreq** settings to match your target hardware and establish an 80MHz clock. For the Hydra these setting would be:*

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

3) Power on and connect your target hardware.

4) Press <F10> to upload and run **PASD_AsmDebugDemo.spin**. *NOTE: Do not close the Propeller IDE.*

5) Double click the **PASD.exe** executable to start it.

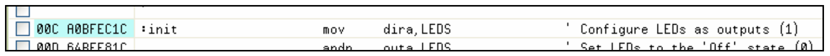
PASD.exe

6) In PASD, Select "Com Port" from the "COM" menu and specify your COM port.

7) Make sure that the Propeller IDE is still open, and that the "**PASD_AsmDebugDemo** spin" window in the Propeller IDE is the currently active window. Press <F2> (or alternatively select "Get Asm Code" from the "File" menu)

   The main PASD window will update to show the source code for the module being debugged (**PASD_AsmDebugDemo.spin).**

8) Note that the current line ($00C) is highlighted in blue. This is the first line which will be run when execution in started.


```
00C A0BFEC1C :init        mov     dira,LEDS          ' Configure LEDs as outputs (1)
00D 64BFE81C              andn    outa,LEDS          ' Set LEDs to the 'Off' state (0)
```

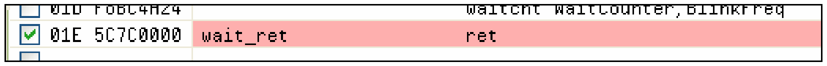
9) Single step the code by pressing <F8> or choosing "Step" from the "Debug" menu. Note that line $00C has executed and the current line is now $00D.

10) Place a breakpoint on line $017 by checking the box at the left of that line.

11) Run the code by pressing <F5> or choosing "Run" from the "Debug" menu. Note that execution has halted on line $017 where the breakpoint was set.

12) Try to set a breakpoint on line $01E by checking the box at the left of that line. Note that the line becomes highlighted in red, and that an error message is displayed at the bottom of the PASD main window.

PASD does not support setting breakpoints on instructions which will be modified at runtime (i.e. the ret instruction, or self-modifying code).



```
   01D F0BC4A24                              waitcnt WaitCounter,BlinkFreq
✓  01E 5C7C0000   wait_ret                   ret
```

13) Select "Clear All Breakpoints" from the "Debug" menu.

14) Open up the Pin Viewer window by selecting "Pin Viewer" from the "Debug" menu.

15) Run the code by pressing <F5> or choosing "Run" from the "Debug" menu. Note that the Pin Viewer window is continuously updated to reflect the state of the Propeller's I/O pins during operation.

16) Stop the code by pressing <F6> or choosing "Stop" from the "Debug" menu. Note that execution has returned to the first line of code ($00C). Since the Propeller does not support interrupts the PASD Debug Kernel cannot regain control of code without a breakpoint being set (Note: single-stepping creates a temporary breakpoint on the next instruction which is immediately removed after execution). If you use "Stop" to halt a program's execution then PASD must unload and reload the COG to regain control, and execution restarts at the first line of your target program.

4. Setting up your spin code to use PASD

PASD can be used to debug a single assembly module running in a single target COG. To set your spin code up to use PASD:

1) Include the "PASDebug" object by placing the following line of code into your target module:

```
OBJ
...
dbg    :         "PASDebug"
```

2) Start the "PASDebug" object during your module initialization by placing the following line of code into your target module:

```
PUB main
...
      dbg.start(31,30,@entry)
```

NOTE: The numbers 31 and 30 are the serial receive and transmit pins respectfully.

3) Insert the 12 long-word PASD Debugger Kernel at the start of your target assembly code. It is critical that the PASD Debugger Kernel be the first instructions in your assembly code.

NOTE: You must make sure that the cognew() call which envokes your assembly code (and the dbg.start() call which starts the debugger) both pass the start address of the Debug Kernel, not to the start address of your target code which follows the Debug Kernel. In the example shown below, the label "entry" is used.

```
DAT

          org     0
entry

'-- Debugger Kernel add this at Entry (Addr 0)
   long $34FC1202,$6CE81201,$83C120B
   long $8BC0E0A,$E87C0E03,$8BC0E0A
   long $EC7C0E05,$A0BC1207,$5C7C0003
   long $5C7C0003,$7FFC,$7FF8
'----------------------------------------
...
```

5. Operation

5.1. Main Application Menus

5.1.1. File Menu

Get Asm Code (F2)

Retrieves the Asm source code from the currently active window of the currently open Propeller IDE.

Upload Code (F11)

Forces the Propeller IDE to upload the module in the currently active IDE window.

Open Source File

Opens an Asm source file from disk.

Font Size

Sets the font size used in the main window.

Save Settings

Saves the PASD configuration. The saved configuration will be loaded the next time PASD is started. The configuration settings include the COM port setting.

Quit

Closes PASD

5.1.2. Debug Menu

Run (F5)

Starts program execution. Execution will halt when a breakpoint is reached.

Stop (F6)

Stops program execution. The target module is restarted, and the program counter returns to the first instruction in the module.

Step (F8)

Executes the current instruction (highlighted in blue). Execution halts on the next instruction.

Step Over (F7)

Executes the current instruction (highlighted in blue). If the instruction is a call, then execution halts on the instruction following the call.

Set Address

Sets the program counter to the selected instruction line. *Note: to select an instruction line, click the address/data field on the left side of the line.*

Toggle Breakpoint (F9)

Toggles a breakpoint on the currently selected instruction line. *Note: to select an instruction line, click the address/data field on the left side of the line.*

Clear All Breakpoints

Removes all currently set breakpoints.

COG RAM Viewer

Opens the COG RAM Viewer window.

Main RAM Viewer

Opens the Main RAM Viewer window.

Pin Viewer

Opens the Pin Viewer Window.

5.1.3. COM Menu

COM Port

Sets the COM port used to communicate with the target hardware.

COM Open/Close

When checked, the COM port is open. When unchecked, the COM port is closed.

Auto Mode

When checked…

- The COM port is opened and closed automatically when an "Upload Code" command is executed.
- The COM port is closed automatically if PASD is minimized
- When PASD is restored (un-minimized) the COM port is opened and the source code is re-loaded from the IDE (if "Get ASM Code" was originally used) or from disk (if "Open Source Code" was originally used).

When unchecked, the COM port may be opened and closed manually using the "COM Open/Close" setting.

5.1.4. Help Menu

The Help menu provides facilities for accessing help, the website, and PASD build information.

5.2. COG RAM Viewer

5.2.1. Viewer Operation

The COG RAM viewer displays the entire contents of COG RAM on the target COG. The COG RAM viewer window is updated every time code execution is halted for a breakpoint or a single-step execution request. Any value which has changed between the time code execution was last started and the time execution was halted will be highlighted in green.

Addr:

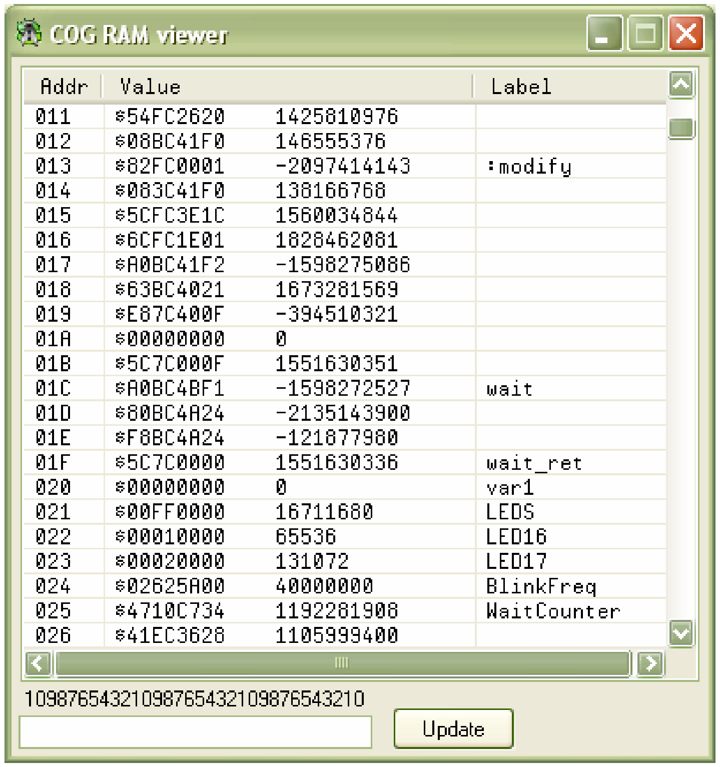
The COG RAM long-word address, in hex.

Value:

The contents of the COG RAM long-word, displayed in both Hex and signed decimal.

Label:

The source code label associated with the COG RAM long-word (if one exists).



Addr	Value		Label
011	$54FC2620	1425810976	
012	$08BC41F0	146555376	
013	$82FC0001	-2097414143	:modify
014	$083C41F0	138166768	
015	$5CFC3E1C	1560034844	
016	$6CFC1E01	1828462081	
017	$A0BC41F2	-1598275086	
018	$63BC4021	1673281569	
019	$E87C400F	-394510321	
01A	$00000000	0	
01B	$5C7C000F	1551630351	
01C	$A0BC4BF1	-1598272527	wait
01D	$80BC4A24	-2135143900	
01E	$F8BC4A24	-121877980	
01F	$5C7C0000	1551630336	wait_ret
020	$00000000	0	var1
021	$00FF0000	16711680	LEDS
022	$00010000	65536	LED16
023	$00020000	131072	LED17
024	$02625A00	40000000	BlinkFreq
025	$4710C734	1192281908	WaitCounter
026	$41EC3628	1105999400	

10987654321098765432109876543210

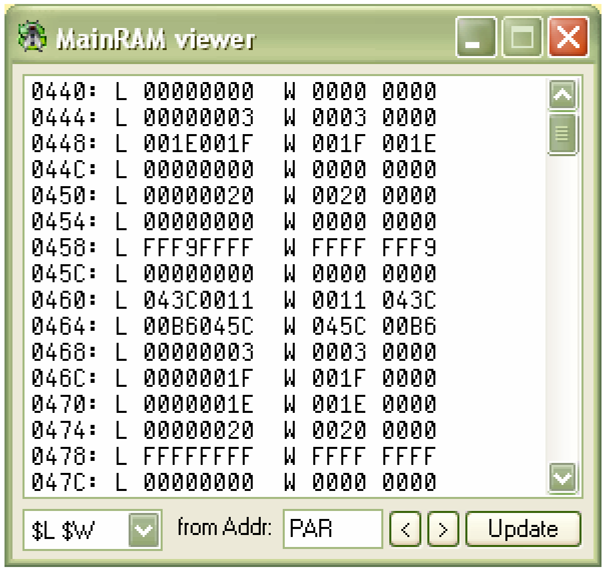
5.2.2. Modifying COG RAM

To modify a COG RAM long-word:

1) Click the "Addr" field of the line containing the COG RAM long-word to be modified.
2) The current long-word's value will be shown in binary inside the edit box at the bottom of the COG RAM viewer. The numbers above the edit box indicate the 32 available bit positions.
3) Click the edit box. The box will be cleared to contain a single equals sign "=".
4) Enter the new value after the equals sign either in decimal with no prefix (e.g. "=1024"), or in hex with a "$" prefix (e.g. =$aa55aa55).
5) Click "Update"

5.3. Main RAM Viewer

The Main RAM viewer displays 128 longwords of main memory starting at the specified base address, and can be used to view both RAM and ROM.



```
0440: L 00000000  W 0000 0000
0444: L 00000003  W 0003 0000
0448: L 001E001F  W 001F 001E
044C: L 00000000  W 0000 0000
0450: L 00000020  W 0020 0000
0454: L 00000000  W 0000 0000
0458: L FFF9FFFF  W FFFF FFF9
045C: L 00000000  W 0000 0000
0460: L 043C0011  W 0011 043C
0464: L 00B6045C  W 045C 00B6
0468: L 00000003  W 0003 0000
046C: L 0000001F  W 001F 0000
0470: L 0000001E  W 001E 0000
0474: L 00000020  W 0020 0000
0478: L FFFFFFFF  W FFFF FFFF
047C: L 00000000  W 0000 0000
```

The data display format can be selected by choosing any of the following options from the drop-down menu.

Mode	Example
`$L $W`	`0470: L $044C0011  W 0011 044C`
`$L dec`	`0470: L $044C0011  dec 72089617`
`$B 'B`	`0470: $B 11 00 4C 04  'B l...`
`%L`	`0470: L %00000100010011000000000000010001`

To change the base address of the memory display, enter the new base address into the edit box, in hex, and press "Update". You can also specify "PAR" to view the contents of Main RAM pointed to by the target COG's PAR (Cog Boot Parameter) register.

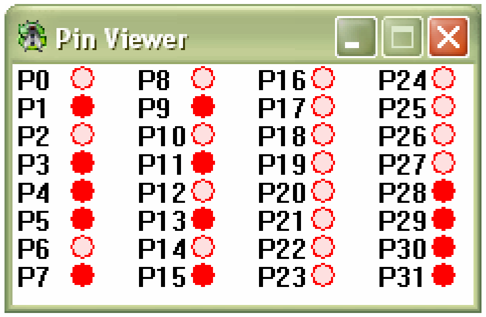
To move backward or forward 128 longwords use the backward "<" or forward ">" button.

5.4. Pin Viewer

The Pin Viewer window shows the current state of all Propeller pins. Pins in a "High" (i.e. "1") state are shown in dark red. Pins in a "Low" (i.e. "0") state are shown in pink.

● High, "1"

○ Low, "0"



P0	○	P8	○	P16	○	P24	○
P1	●	P9	●	P17	○	P25	○
P2	○	P10	○	P18	○	P26	○
P3	●	P11	●	P19	○	P27	○
P4	●	P12	○	P20	○	P28	●
P5	●	P13	●	P21	○	P29	●
P6	○	P14	○	P22	○	P30	●
P7	●	P15	●	P23	○	P31	●

The pin viewer window is updated every time code execution is halted for a breakpoint or a single-step execution request, and is updated continuously whenever code is running (i.e. when the "Run" command has been issued, and a breakpoint has not yet been reached).

5.5. Keyboard Shortcuts

The following keyboard commands can be used to control PASD:

Key	Function
F2	Get Asm Code
F5	Run
F6	Stop
F7	Step Over
F8	Step
F9	Toggle Breakpoint
F11	Upload Code

6. Under The Hood

The Propeller chip does not contain hardware to specifically support breakpoints. Instead, it is necessary to simulate breakpoints by temporarily modifying target instructions.

When you set a breakpoint, PASD replaces the target instruction with a jump to the PASD debugging kernel, and stores the original instruction in the Windows PASD Application. Whenever you clear a breakpoint, the original instruction is restored.

In the main PASD source-code window you will always be shown the "orginal" source code instruction even though it may have been temporarily replaced. In the PASD COG RAM Viewer you will be able to see the substituted jump instruction, and the line will be highlighted in red to indicate that it is a breakpoint.

PASD is not able to support breaking on instructions which are modified at run-time (such as the "RET" instruction) because in order to establish the breakpoint the instruction has to be temporarily replaced with a jump, and the run-time instruction modification would be made to the *jump* instead of the intended *original* instruction, which would corrupt the jump and cause unpredictable operation.

Single step execution for opcodes other than JUMP is simulated by copying the next instruction to the PASD Kernel and executing it there. Single step execution for JUMP opcodes is performed by setting two temporary breakpoints; one at the instruction following the JUMP and one at the JUMP location. This is necessary to support stepping through either branch of a conditional JUMP.

7. Limitations

7.1. Number of COGs

PASD currently only supports debugging a single assembly module executing in a single COG.

7.2. Runtime Modified Instructions

PASD cannot generally support breakpoints on instructions which may be programmatically modified. This includes self-modifying code, and the "RET" instruction.

You can let PASD know that an instruction will be modified at runtime by using "0-0" as the instruction's source or destination field (or both). If you attempt to set a breakpoint on an instruction designated "0-0" PASD will warn you and highlight the line in red.

The Propeller architecture is designed to support the RET instruction by modifying the return address contained in the RET opcode whenever it's associated CALL or JMPRET opcode is executed. If you attempt to set a breakpoint on a RET instruction PASD will warn you and highlight the line in red.

7.3. WAITCNT and WAITVID

PASD ignores (i.e. does not execute) WAITCNT and WAITVID instructions when single stepping for two reasons:

1) These commands are meant to synchronize precise timings and are not applicable when single stepping.
2) The system counter (CNT) runs freely when PASD is single stepping even though the target code is "halted" between steps, so executing a WAITCNT instruction would very likely cause the Propeller to wait until the specified count value (if missed) came around again, which could take about 50 seconds.

7.4. Repeated Definitions

Repeated long/word/byte definitions like:

```
long  value[size]
```

are not currently recognized by the parser, and cannot be used with PASD.

8. About Insonix

Insonix specializes in both hardware and software development

We have many years of experience developing electronics. Our main areas of expertise are digital, microprocessor and DSP based designs for the consumer and medical technology industries.

After gathering your ideas and product requirement specifications we provide placement, routing, layout, board design, software and prototypes (and sometimes small production runs). Our strength lies in finding the simplest and most favorable solution in the shortest time, which minimizes development costs.