ZOG: GNU C/C++ for the Propeller

January 28, 2011

Manual by:

David Betz
Michael Rychlik

Software by:

Michael Rychlik
David Betz
Steve Denson

Table of Contents

INTRODUCTION

1.1 What is Zog?

Zog enables running C and C++ programs compiled with GCC, the GNU Compiler Collection, on the Propeller multi-core micro-controller from Parallax Inc.

There is no Propeller architecture target for GCC. To overcome this Zog provides a virtual machine implementing the instruction set of a CPU architecture for which there is a GCC target, namely the Zylin CPU (ZPU) from Zylin Consulting. The ZPU architecture provides a 32-bit processor with a very minimal instruction set, each ZPU instruction is a single byte, it is therefore a perfect candidate for emulation on the Propeller as the entire bytecode interpreter can be implemented in PASM within a single COG. Hence the name ZOG, ZPU in a Cog. Further, as each instruction is a single byte the ZPU is a good match for Propeller systems having external RAM or ROM that is connected over an 8 bit data bus.

Zog can be used to run compiled C, C++ binaries on a bare Propeller where the binaries are located within the Propellers HUB RAM space. This limits the binary size to a little less than 32KBytes. Fortunately there are Propeller systems available that provide memory external to the Propeller chip and Zog can also make use of such memory expansion from which it can execute much larger binaries and deal with much larger data sets. Up to 32MBytes currently!

This document describes how get C and C++ programs running on some of those Propeller platforms with the Zog virtual machine.

If you are interested in the details of the ZPU architecture and instruction set you will find it on the Zylin Consultants WEB site:

http://opensource.zylin.com/zpu.htm

1.2 Supported Boards

This version of ZOG supports the following boards:

1) Parallax Propeller C3 board
2) Gadget Gangster Propeller Platform board with the SDRAM module. To use TV output and PS2 keyboard input requires the SDRAM/TV module.

1.3 Installing ZOG

This release only runs under Windows. Since you are reading this file you must have unzipped the zog-cygwin.zip file successfully. This should have created a directory called *zog-cygwin.* Open up a Windows Command prompt and *cd* to the *zog-cygwin* directory.

Then, edit the *setenv.bat* file in this directory so that the **ZOGPATH** environment variable points to the directory where this *README.txt* file is located and set the **PORT** variable to the COM port that talks to your Propeller board. The **ZOGPATH** variable must be set using the Cygwin path syntax.

In other words, if you've unzipped the zog-cygwin.zip file in the directory

```
C:\Users\myname
```

then you should set the ZOGPATH environment variable like this:

```
set ZOGPATH=/cygdrive/c/Users/myname/zog-cygwin
```

In other words,

```
C:\
```

becomes

```
/cygdrive/c/
```

and the rest of the path stays the same with the exception of converting all backslashes to forward slashes. If you've installed on a different drive the procedure is similar.

For instance,

```
D:\
```

becomes

```
/cygdrive/d/
```

You should also change the **BOARD** environment variable to reflect the Propeller board you will be using. Select *c3* for the Parallax Propeller C3 board and *sdram* for the Gadget Gangster Propeller Platform with SDRAM board. There is an additional *c3ram* board type that you can use if you want to run code from the C3 SPI SRAM rather than the C3 SPI flash.

Once you’ve completed editing your *setenv.bat* file type *setenv* at the Windows command prompt to run the *setenv* batch file.

This should setup your environment to work with your Propeller board.

2 Sample Programs

You can get a feel for writing ZOG C programs by looking at the examples in the *tests* directory.

The *hello* test program requires a TV and PS2 keyboard to be connected to your Propeller board. The *filetest, setjmp,* and *xbasic* test programs only require the USB serial connection that you use to download programs to your board.

2.1 hello

To build the *hello* test program, *cd* to the *tests\hello* directory and type *make.* This should run the ZPU C compiler to compile and link the *hello* test program. You will find the resulting program in *tests\hello\bin\hello.zbn.* Files with the extension *.zbn* are ZOG binary images that are ready to load into your Propeller board. To load the *hello* test program, type *make run.* This should load the *hello* test program into your board and start it running. You should see semi-interesting stuff on the TV you have plugged into the video output. Once you see the “Hello, world!” banner line and the display of the *CLKFREQ* and *CNT* registers, you can use your PS2 keyboard to type characters that should be echoed to the TV.

2.2 filetest

To build the *filetest* test program, *cd* to the *tests\filetest* directory and type *make.* This should run the ZPU C compiler to compile and link the *filetest* test program. You will find the resulting program in *tests\filetest\bin\filetest.zbn.* Files with the extension *.zbn* are ZOG binary images that are ready to load into your Propeller board. To load the *filetest* test program, type *make run.* This should load the *filetest* test program into your board and start it running. To run the *filetest* program you will need to insert a formatted SD card into your board. The *filetest* program just writes two files, *test.dat* and *test2.dat* to the SD card and reads them back to verify that they were written correctly. It currently only works on the *Propeller C3* board.

2.3 xbasic

To build the *xbasic* test program, *cd* to the *tests\xbasic* directory and type *make.* This should run the ZPU C compiler to compile and link the *xbasic* test program. You will find the resulting program in *tests\xbasic\bin\xbasic.zbn.* Files with the extension *.zbn* are ZOG binary images that are ready to load into your Propeller board. To load the *xbasic* test program, type *make run.* This should load the *xbasic* test program into your board and start it running. The *xbasic* program is a simple line-oriented Basic interpreter that uses the serial port for input/output. You can try it by typing this simple program:

```
10 for x=1 to 10
```

```
20 printf(“%d %d\n”, x, x*x)
30 next x
```

You can use the *list* command to list the program and the *run* command to run the program. When you run it you will see it print the numbers for 1 to 10 and their squares.

3 Building Programs

3.1 Board Configurations

1) c3 – this should be used with the Parallax Propeller C3 board when you want to compile code that loads into the 1mb SPI flash. This allows up to 1mb of code and 64k of data using both the SPI flash and SPI SRAM chips on the C3.
2) c3ram – this should be used with the Parallax Propeller C3 board when you want to compile and run code that loads into the SPI SRAM chips. This allows up to 64k of code and data and leaves the SPI flash chip alone for other uses.
3) sdram – this should be used with the Gadget Gangster Propeller Platform board with the SDRAM module.

3.2 Make Files

common.mk

The test programs are built using the *make* utility and the *include\common.mk* makefile fragments. You should be able to look at the makefiles in the *tests/hello* and *tests/filetest* directories for an example of how to write your own makefiles to build simple C programs for ZOG. For example, here is the makefile for *hello:*

```
TARGET=hello
SOURCES=hello.c initio.c
include $(ZOGPATH)/include/common.mk
```

The *TARGET* variable specifies the name of the program to build. The resulting ZOG compile image will have this name with *.zbn* appended.

The *SOURCES* variable is a list of source files separated by spaces. If you have more source files than will fit on a line, you can continue them on multiple lines by ending each line with a space followed by a backslash. There should be no space or any other characters after the backslash. For example:

```
SOURCES=hello.c \
source2.c \
source3.c \
initio.c
```

The last line should not end in a backslash.

The final line of the *hello* makefile includes the *common.mk* makefile fragments that define how ZOG programs should be built.

For a more complex example with multiple source files look at *tests/xbasic/makefile*.

You can also invoke the ZPU tools directly without using *make.* Look at *common.mk* to see which options are needed for the ZPU tools and be sure to include the correct linker script for the board configuration you are targeting. The linker scripts have the same names as the board configurations with *.ld* appended.

4 ZOGLOAD

4.1 Description

ZOGLOAD is a program that runs on the PC under either Windows or Linux and provides a way to download files to any of the supported boards. It can write compiled C programs either to memory or an SD card. Programs compiled for the *c3* board configuration can be downloaded to flash. Programs compiled for the *c3ram* or *sdram* board configurations can be downloaded into RAM.

4.2 Selecting the Port (-p)

You select which communications port is used to talk to your Parallax C3 board using the *-p* option. You should follow the -p with the communications port identifier. On Windows this can be either *COMn* or just *n.* If you don't specify a port on the zogload command line the default is taken from the *PORT* environment variable. This is usually setup by the *setenv.bat* file described in the section on *Installing ZOG.*

Example

```
zogload -p COM3
```

or

```
zogload -p3
```

4.3 Selecting the Board Configuration (-b)

You select your board configuration by using the *-b* option. The two choices for board configuration are *c3* for programs intended to run from SPI flash on the C3 and *c3ram* for programs intended to run from SPI SRAM. The *sdram* board configuration is for the Propeller Platform board with SDRAM module. If you don't specify a board configuration on the zogload command line the default is taken from the *BOARD* environment variable. This is usually setup by the *setenv.bat* file described in the section on *Installing ZOG.*

Example

```
zogload -b c3
```

or

```
zogload -b c3ram
```

or

```
zogload –b sdram
```

4.4 Writing a File to Flash or RAM (-wm)

You write a program into memory by using the *-wm* option. This option should be followed by the name of the compiled ZOG program to write. Programs compiled using the *c3* board configuration will be loaded into flash, programs compiled with the *c3ram* or *sdram* board configurations will be loaded into RAM. The *-wm* option is the default if no option is specified before a filename so it can be left out if you want to load a program into memory.

Example

```
zogload –wm hello.zbn
```

or

```
zogload hello.zbn
```

4.5 Writing a File to the SD Card (-wf)

Example

```
zogload -wf hello.zbn
```

4.6 Setting the Program Name (-n)

The *–n* option is used to specify a name to be used by a subsequent option. It is usually used with the *–wb* to write the *zogload.run* file or with the *–r* option to run a program from the SD card.

Example

```
zogload -n hello.zbn -r
```

4.7 Writing a Bootloader to the EEPROM (-we)

There are two bootloaders that zogload can write to the EEPROM. The *sd* bootloader mounts the SD card and looks for a file named *zogload.run* in the root directory. It reads that file and uses its contents as the name of the file to load. For example, if *zogload.run* contains the string “hello.zbn” that file will be loaded and started automatically on reset.

Example

```
zogload –we sd
```

The other bootloader is *flash.* It loads and starts whatever program is programmed in the flash memory. This is the last program written to flash using the *–wm* command with a program compiled for a flash board configuration, currently only *c3.*
Example

```
     zogload -we flash
```

4.8 Setting the File to be Loaded by the Bootloader (-wb)

If you’ve programmed the *sd* bootloader into EEPROM you will need to write the *zogload.run* file to the SD card so the bootloader knows which ZOG program to run on reset. You do this with the *–wb* option. The *–wb* option writes the name of the most recently mentioned file to *zogload.run* on the SD card.

Example

```
     zogload –wf hello.zbn –wb
or
     zogload –n hello.zbn -wb
```

4.9 Running a Program (-r)

The *–r* option runs a previously loaded program. On a board that supports flash memory, it can run the program that is in the flash using just the *–r* option itself.

Example

```
     zogload –r
```

In all other cases, there must be a file to run specified on the command line. This can be done in several ways. The file could have just been loaded into memory with the *–wm* option.

Example

```
     zogload -wm hello.zbn –r
```

Or, the file could have just been written to the SD card with the *–wf* option.

Example

```
     zogload –wf hello.zbn –r
```

Or lastly, the file could already be on the SD card. In that case the *–n* option can be used.

Example

```
     zogload –n hello.zbn –r
```

4.10 Terminal Emulation (-t)

Any of the commands can be followed by *–t* to enter terminal mode after the load is complete. This is a simple terminal emulator that sends the characters typed on the PC keyboard to the Propeller's serial port on pins 30/31 and displays characters transmitted by the Propeller on the terminal.

Example

```
zogload -wm hello.zbn -r -t
```

5 C Runtime Library

5.1 Initialization

Many C programs interact with the user by reading text from *stdin* and writing output to *stdout* and *stderr.* The ZOG runtime library (based on newlib) supports this simple terminal I//O in a number of different ways. By default, because embedded programs often don't have a text interface, the *stdin/stdout/stderr* files are connected to */dev/null.* In other words, reads from *stdin* will always return EOF and writes to *stdout* or *stderr* will just be ignored.

However, it is sometimes useful to be able to use terminal I/O, especially when porting generic C programs to the Propeller. You can arrange to use terminal I/O by including in your program the function *_initIO.* This function is called by the C startup code before your *main* function is called. This allows you to setup terminal I/O. One advantage of using *_initIO* is that you don't have to modify generic C code to make it work on the Propeller. You just have to add an additional module containing your definition of the *_initIO* function and link it with the generic C source files.

Here is an example of a simple *_initIO* function that sets up serial terminal I/O:

```
#include "propeller.h"

void _initIO(void)
{
    InitSerialTerm(
          conRxPin,
          conTxPin,
          conMode,
          conBaud,
          TERM_IO);
}
```

The *conRxPin, conTxPin, conMode,* and *conBaud* values are defined in the *include/board_xxx.h* file associated with the board configuration you have chosen. For instance, for the *c3* board configuration, they are in the file *include/board_c3.h.* The *TERM_IO* parameter says that both input and output will be directed to the serial port.

Here is another example to setup TV output and PS2 keyboard input

```
#include "propeller.h"

void _initIO(void)
{
    InitKeyboardTerm(keybdPinD, keybdPinC);
    InitTvTerm(tvPin);
}
```

Again, *keybdPinD, keybdPinC,* and *tvPin* are defined in the relevant *board_xxx.h* file.

5.2 Serial I/O

```
int InitSerialTerm(
          int rxpin,
          int txpin,
          int mode,
          int baudrate,
          int flags);
```

You can use this function to setup terminal I/O on a serial port. You can setup either input or output or both. It loads the FdSerial driver into a COG and directs *stdin*, *stdout*, and *stderr* to the serial port. The *rxpin, txpin, mode,* and *baud* parameters are the same as the corresponding parameters to *FdSerial_start.*

Example

```
InitSerialTerm(31, 30, 0, 115200, TERM_IO);
```

Use TERM_IO to setup both serial input and serial output for *stdin*, *stdout*, and *stderr*. Use TERM_IN to setup just serial input from *stdin* and TERM_OUT to setup just serial output to *stdout* and *stderr*.

5.3 Keyboard Input

```
int InitKeyboardTerm(
          int dpin,
          int cpin);
```

You can use this function to setup terminal input from a PS2 keyboard for *stdin*. It loads the Keyboard driver into a COG and directs *stdin* to the keyboard. The *dpin* and *cpin* parameters are the same as the corresponding parameters to *keybd_start.*

Example

```
InitKeyboardTerm(26, 27);
```

5.4 TV Output

```
int InitTvTerm(
          int tvpin);
int InitTv2Term(
          int tvpin);
```

You can use these functions to setup terminal output to the TV for *stdout* and *stderr*. The *InitTvTerm* function uses the Parallax *TV* driver that supports 42 columns by 14 rows. The *InitTv2Term* function uses the *TV_Half_Height* driver that supports 44 columns by 54 rows. The *tvpin* parameter is the same as the corresponding parameter to *tvText_start* and *tv2Text_start*.

Example

```
InitTvTerm(12);
```

5.5 File I/O

```
int InitC3FileIO(
          int retries);
```

You can use this function to enable SD card file I/O when using the *c3* or *c3ram* board configurations. The *InitC3FileIO* function mounts the SD card and redirects all file I/O to the SD card. The *retries* parameter determines how many times to attempt to mount the SD card. Specify zero for *retries* to try indefinitely. The C3 file I/O driver uses the same COG that is used by ZOG to manage the SPI flash and SRAM memories. No additional COG is loaded by this function.

Example

```
InitC3FileIO(10);
```

6 Drivers

The following drivers have been ported to ZOG. Check the header files for them in the *include* directory for more information on the functions that are available.

6.1 FdSerial

#include “FdSerial.h”

6.1.1 FdSerial_start

```
int FdSerial_start(
     FdSerial_t *data,
     int rxpin,
     int txpin,
     int mode,
     int baudrate);
```

Initializes and starts a serial driver in a COG.

6.1.2 FdSerial_stop

```
void FdSerial_stop(
     FdSerial_t *data);
```

Stops a serial driver.

6.1.3 FdSerial_rxflush

```
void FdSerial_rxflush(
     FdSerial_t *data);
```

Empties the receive queue.

6.1.4 FdSerial_rxcheck

```
int FdSerial_rxcheck(
     FdSerial_t *data);
```

Gets a byte from the receive queue if one is available. Otherwise, it returns -1.

6.1.5 FdSerial_rxtime

```
int FdSerial_rxtime(
     FdSerial_t *data,
     int ms);
```

Gets a byte from the receive queue if one is available before the specified timeout in milliseconds. Otherwise, it returns -1 to indicate timeout.

6.1.6 FdSerial_rx

```
int FdSerial_rx(
     FdSerial_t *data);
```

Waits for a byte from the receive queue.

6.1.7 FdSerial_tx

```
int FdSerial_tx(
     FdSerial_t *data,
     int txbyte);
```

Puts a byte into the transmit queue.

6.2 TvText

The following functions are provided in the TvText driver. For more information, see *include/TvText.h.*

```
#include “TvText.h”

int tvText_start(int basepin);
void tvText_stop(void);
void tvText_str(char* sptr);
void tvText_dec(int value);
void tvText_hex(int value, int digits);
void tvText_bin(int value, int digits);
void tvText_out(int c);
void tvText_setColorPalette(char* palette);
uint16_t tvText_getTile(int x, int y);
void tvText_setTile(int x, int y, uint16_t tile);
uint16_t tvText_getTileColor(int x, int y);
void tvText_setTileColor(int x, int y, uint16_t color);
void tvText_setCurPosition(int x, int y);
void tvText_setCoordPosition(int x, int y);
void tvText_setXY(int x, int y);
void tvText_setX(int value);
void tvText_setY(int value);
int tvText_getX(void);
int tvText_getY(void);
void tvText_setColors(int value);
int tvText_getColors(void);
int tvText_getColumns(void);
int tvText_getRows(void);
```

6.3 Tv2Text

The following functions are provided in the Tv2Text driver. For more information, see *include/Tv2Text.h.*

```
#include “Tv2Text.h”

int tv2Text_start(int basepin);
void tv2Text_stop(void);
void tv2Text_str(char* sptr);
void tv2Text_dec(int value);
void tv2Text_hex(int value, int digits);
void tv2Text_bin(int value, int digits);
void tv2Text_out(int c);
void tv2Text_setColorPalette(char* palette);
uint16_t tv2Text_getTile(int x, int y);
void tv2Text_setTile(int x, int y, uint16_t tile);
uint16_t tv2Text_getTileColor(int x, int y);
void tv2Text_setTileColor(int x, int y, uint16_t color);
void tv2Text_setCurPosition(int x, int y);
void tv2Text_setCoordPosition(int x, int y);
void tv2Text_setXY(int x, int y);
void tv2Text_setX(int value);
void tv2Text_setY(int value);
void tv2Text_setYhalf(int value);
int tv2Text_getX(void);
int tv2Text_getY(void);
void tv2Text_setColors(int value);
int tv2Text_getColors(void);
int tv2Text_getColumns(void);
int tv2Text_getRows(void);
void tv2Text_button(
     int left,
     int top,
     int width,
     int height,
     int color,
     char* text);
void tv2Text_box(
     int left,
     int top,
     int width,
     int height,
     int color);
void tv2Text_rectangle(
     int left,
     int top,
     int width,
     int height,
     int color,
     int fill);
```

6.4 TvTileGfx

```
#include “TvTileGfx.h”
```

6.4.1 TvTileGfx_start

```
int TvTileGfx_start(
     TvTileGfx_st *data,
     int basepin);
```

Initializes and starts the TV tile graphics driver in a COG.

6.4.2 TvTileGfx_stop

```
void TvTileGfx_stop(
     TvTileGfx_st *data);
```

Stops the TV tile graphics driver. Before calling this function the first four fields of the TvTileGfx_st structure should be setup. The first three must point to buffers in hub memory.

6.5 Keyboard

```
#include “Keyboard.h”
```

6.5.1 keybd_start

```
int keybd_start(
     int dpin,
     int cpin);
```

Initializes and starts the keyboard driver in a COG.

6.5.2 keybd_startx

```
int keybd_startx(
     int dpin,
     int cpin,
     int locks,
     int autorep);
```

Like start, but allows you to specify lock settings and auto-repeat.

6.5.3 keybd_stop

```
void keybd_stop(void);
```

Stops the keyboard driver.

6.5.4 keybd_present

```
int keybd_present(void);
```

Checks to see if a key is present.

6.5.5 keybd_key

```
int keybd_key(void);
```

Gets a key from the buffer and returns zero if the buffer is empty.

6.5.6 keybd_getkey

```
int keybd_getkey(void);
```

Gets a key and waits if none is currently available.

6.5.7 keybd_newkey

```
int keybd_newkey(void);
```

Clears the buffer and waits for a new key to arrive.

6.5.8 keybd_gotkey

```
int keybd_gotkey(void);
```

Checks to see if there are any keys in the buffer. (How is this different from keybd_present?)

6.5.9 keybd_clearkeys

```
void keybd_clearkeys(void);
```

Clears the key buffer.

6.5.10 keybd_keystate

```
int keybd_keystate(
     int key);
```

Get the state of a key.

7 Propeller Interface

7.1 Hub Memory Variables

Sometimes it is necessary to place variables in hub memory so that they can be accessed by another COG. You can do this by using the special *HUB* memory attribute that is defined in *propeller.h*.

Example

```
#include “propeller.h”
HUB uint32_t params[2];
```

This will place the *params* variable in hub RAM.

7.2 Memory Map

7.2.1 C3 Memory Map

```
0x00000000 - 0x0000ffff  Cached external RAM
0x00100000 - 0x001fffff  Cached external ROM
0x10000000 - 0x10007fff  Hub RAM
0x10008000 – 0x1000ffff  Hub ROM
0x18000000 – 0x100081ff  COG memory
```

7.2.2 Propeller Platform SDRAM Memory Map

```
0x00000000 - 0x0fffffff  Cached external memory
0x10000000 - 0x10007fff  Hub RAM
0x10008000 – 0x1000ffff  Hub ROM
0x18000000 – 0x100081ff  COG memory
```

7.3 Propeller Registers

There are a number of the Propeller registers that are available from C code.

7.3.1 Hub Variables

```
CLKFREQ
```

CLKFREQ isn’t a register. It is a location in hub memory that is initialized with the current clock frequency.

7.3.2 COG Registers

The following registers are available from C code. Keep in mind that these are the registers of the COG that is running the ZOG virtual machine. For a description of each register see the Propeller Manual.

```
PAR
CNT
INA
INB
```

```
OUTA
OUTB
DIRA
DIRB
FRQA
FRQB
PHSA
PHSB
VCFG
VSCL
```

7.4 Propeller Functions

7.4.1 cogid

```
int cogid(void);
```

Return the ID of the current COG.

7.4.2 cognew

```
int cognew(
     int *code,
     int size,
     int *par);
```

Allocate a COG, load *code* into it and start it running passing it *par* as a parameter. The *size* parameter gives the size of the code in longs. If *par* is a pointer it must point to hub memory. The *code* need not be in hub memory.

7.4.3 coginit

```
int coginit(
     int cog,
     int *code,
     int size,
     int *par);
```

Load *code* into *cog* and start it running passing it *par* as a parameter. The *size* parameter gives the size of the code in longs. If *par* is a pointer it must point to hub memory. The *code* need not be in hub memory.

7.4.4 cogstop

```
void cogstop(
     int cog);
```

Stop *cog* from running and make it available again for allocation by *cognew.*

7.4.5 longfill

```
void longfill(
     uint32_t *dst,
     uint32_t fill,
     int count);
```

Fill memory at *buf* with *count* 32 bit *fill* values.

7.4.6 longmove

```
void longmove(
     uint32_t *dst,
     uint32_t *src,
     int count);
```

Copy *count* 32 bit values from *src* to *dst.*

7.4.7 longmove_swapwords

```
void longmove_swapwords(
     uint32_t *dst,
     uint32_t *src,
     int count);
```

Copy *count* 32 bit values from *src* swapping the words in each value before writing it to *dst.*

8 Summary

This is a brief summary of the features of ZOG. Please refer to the included source code and examples for more information.