Propeller Loader

by David Betz and Steve Denson

Julu 20, 2012

1 Table of Contents

2 Introduction

The Propeller Loader (propeller-load) is a command line program used to load programs generated by the PropGCC toolchain into a Propeller board. It loads over a serial connection from a PC running Windows, Mac OS X, or Linux. It can also load Spin binary programs generated by programs like the Propeller Tool or BST.

3 Common Use Cases

The propeller-load program uses the Propeller chip's built-in loader to load LMM (-mlmm) and COG (-mcog) mode programs. This means that it can be used to load programs to a board that doesn't have a crystal since the built-in loader can handle the imprecise timing obtained when using the RCFAST clock mode. However, to load XMM (-mxmmc, -mxmm-single, or –mxmm-split) programs, propeller-load uses a secondary loader that requires more precise timing and can only operate reliably on a board with a crystal. This is also true of programs that make use of drivers placed in EEPROM (.ecog files) above 32k.

3.1 Loading COG or LMM Programs

Loading a COG (-mcog) or LMM (-mlmm) mode program is done in a single stage using the Propeller chip's boot loader. The only board configuration parameters that are used for this type of load are the *baudrate, clkfreq,* and *clkmode* settings. The –b option can be omitted if the *default* board configuration is adequate. The default configuration assumes an 80mhz clock with clock mode XTAL1+PLL16X and a baud rate of 115200. However, if the program being loaded makes use of variable patching as described later in this document the board type should be specified in the load command. In general, it's best to always specify the board type.

Examples

```
propeller-load –b c3 myprog.elf –r –t
```

This command loads the program *myprog.elf,* starts it running, and then enters the terminal emulator.

```
propeller-load –b c3 myprog.elf –e –r -t
```

This command loads the program myprog.elf, writes it to the EEPROM, starts it running, and then enters the terminal emulator.

3.2 Loading an XMM Program

Loading an XMM (-mxmmc, -mxmm-single, or –mxmm-split) program is done in two stages. The first stage uses the Propeller chip's boot loader to load a helper program that contains a driver that knows how to write into the target board's external memory. In the second stage, the loader talks to this helper program to load the XMM program into external memory. A board type is always required for XMM loads since the board configuration file contains the name of the driver to use to access external memory. This is called the cache driver and it is also used once the XMM program is running to allow the XMM kernel to access external memory.

Examples

```
propeller-load –b c3 myprog.elf –r –t
```

This command loads the program *myprog.elf* into external memory starts it running, and then enters the terminal emulator.

```
propeller-load –b c3 myprog.elf –e –r -t
```

This command loads the program myprog.elf into external memory, writes a flash loader to the EEPROM, starts the program running, and then enters the terminal emulator. The –e option only makes sense when external flash memory is available on the target board. This is because the loader that is written to EEPROM assumes it will find the XMM program in external memory and that requires that at least the portion of the external memory that contains code be non-volatile. The –e option can be used on a board that has both external flash and SRAM if either the –mxmmc or –mxmm-split memory models are used since in those models the code is written to the flash.

3.2.1 Cache Drivers

Cache drivers are used by XMM programs to write the program to external memory during loading and also to access external memory at runtime. There is usually at least one dedicated cache driver for each type of target board although some generic cache drivers exist that will work with any board with specific memory parts. These generic cache drivers include an EEPROM cache driver that works with boards that have a single EEPROM of 64k or larger, an SD cache driver that works with any board with an SD card slot, and drivers that work with boards that have specific types of SPI flash or SRAM chips.

3.3 Using the SD Loader

The SD loader provides a way to load XMM programs from an SD card. This is mostly useful for boards that have external RAM but not flash.

Examples

```
propeller-load –b c3 –l myprog.elf –r –t
```

This command writes the program *myprog.elf* to the SD card as *autorun.pex,* loads a helper program that loads the program from the SD card into external memory, starts the helper program running, and then enters the terminal emulator.

```
propeller-load –b c3 –l myprog.elf –e –r -t
```

This command writes the program *myprog.elf* to the SD card as *autorun.pex,* writes a helper program that loads the program from the SD card into external memory into the EEPROM, starts the helper program running, and then enters the terminal emulator.

```
propeller-load –b c3 –l –r –t
```

This command loads a helper program that loads the program from the file *autorun.pex* found in the root directory of the SD card into external memory, starts the helper program running, and then enters the terminal emulator. To use this command the file *autorun.pex* must already be in the root directory of the SD card.

```
propeller-load –b c3 –l –e –r -t
```

This command writes a helper program that loads the program from the file *autorun.pex* found in the root directory of the SD card into external memory into the EEPROM, starts the helper program running, and then enters the terminal emulator. To use this command the file *autorun.pex* must already be in the root directory of the SD card.

With any of these commands the user can move the SD card to a PC and replace the *autorun.pex* file on it and then move it back to the Propeller board. This can avoid lengthy load times for large programs.

3.4 Using the SD Cache Driver

The SD cache driver provides a way to run XMMC programs using an SD card as external memory.

Examples

```
propeller-load –b c3 –z myprog.elf –r –t
```

This command writes the program *myprog.elf* to the SD card as *autorun.pex,* loads a helper program that prepares to run the program directly from the SD card, starts the helper program running, and then enters the terminal emulator.

```
propeller-load –b c3 –z myprog.elf –e –r -t
```

This command writes the program *myprog.elf* to the SD card as *autorun.pex,* writes a helper program that prepares to run the program directly from the SD card to EEPROM, starts the helper program running, and then enters the terminal emulator.

```
propeller-load –b c3 –z –r –t
```

This command loads a helper program that prepares to run the program from the file *autorun.pex* found in the root directory of the SD card directly from the SD card, starts the helper program running, and then enters the terminal emulator. To use this command the file *autorun.pex* must already be in the root directory of the SD card.

```
propeller-load –b c3 –z –e –r -t
```

This command writes a helper program that prepares to run the program from the file *autorun.pex* found in the root directory of the SD card directly from the SD card into the EEPROM, starts the helper program running, and then enters the terminal emulator. To use this command the file *autorun.pex* must already be in the root directory of the SD card.

3.5 Writing a File to the SD Card

Sometimes it is necessary to write files to an SD card inserted into a Propeller board. An example of this is writing the file *autorun.pex* to the root directory of the SD card to allow its use with the propeller-load –l or –z options. You might also need to write data files to the SD card that will be needed by a program you intend to run.

Example

```
propeller-load –b c3 –f autorun.pex
```

This command writes the file *autorun.pex* to the root directory of the SD card.

3.6 Creating a PEX file

In order to load a program from the SD card or run it directly from the SD card the program must be converted to .pex format. You can do this using the propeller-load –x option.

Example

```
propeller-load –x myprog.elf
```

This command writes the file *myprog.pex* to the same directory that contains *myprog.elf*. You can then write file *myprog.pex* to the SD card with the name *autorun.pex* in order to use the propeller-load –l to load it from the SD card or the –z option to run it directly from the SD card.

3.7 Creating a Spin Binary File

In order to create a Spin binary that you can load using the Propeller Tool or some other loader that uses the Spin binary format from a COG or LMM program, you can use the propeller-load –s option.

Example

```
propeller-load –s myprog.elf
```

This command writes the file *myprog.binary* to the same directory that contains *myprog.elf*..

4 Options

4.1 -b <type> Select target board

Use this option to select the target board type. This determines which board configuration file is used. If this option is not specified, the value of the environment variable *PROPELLER_LOAD_BOARD* is used. If that environment variable is not defined, the "default" board type is used. The configuration file for the selected board type is located by looking in the following places and choosing the first matching file found.

- the directory containing the file being loaded
- the directory given in the environment variable *PROPELLER_LOAD_PATH*
- the directory containing the propeller-load program
- the directory /opt/parallax/propeller-load

Example

```
-b c3
```

This will select the *c3.cfg* board configuration file.

4.2 -D <var>=<value> Define a board configuration variable

Use this option to define or redefine a configuration variable. The loader will use values set using the –D option instead of the corresponding values from the selected board configuration file. The –D option can also be used to define new variables that are not in the board configuration file. This could be useful if the program being loaded makes use of the values of these additional variables through the loader's variable patching facility.

Example

```
-D baudrate=9600
```

This will use 9600 as the baud rate overriding the value for *baudrate* in the selected configuration file.

4.3 -e Write the program into EEPROM

Use this option to write the program being loaded to EEPROM. This option works differently depending on the type of program being loaded.

If a COG or LMM program is being loaded, the entire program is written to the EEPROM.

If an XMM program is being loaded, only a loader is written to EEPROM. The program itself is written to external memory. This requires that the external memory be non-volatile, for example, flash memory.

4.4 -f Write a file to the SD card

Use this option to write a file to an SD card inserted in the target board. If this option is given then no program is loaded. The only action is to write a file to the SD card.

Example

```
-f myprog.pex
```

4.5 -I <path> Add a directory to the include path

Use this option to add a directory to the include path. This is the path that the loader uses to locate board configuration files and drivers. The directories specified using the –I option will be searched before the standard directories.

Example

```
-I foo/bar
```

4.6 -l Write a program to the SD card and use the SD loader

Use this option to load code that can load the program *autorun.pex* from the root directory on the SD card into external memory. If a filename is given in the command, that file will be written to the SD card as *autorun.pex.* It should be an XMM program.

Examples

```
-l
```

Load code to load the program *autorun.pex* which should already be on the SD card

```
-l myprog.elf
```

Write the program myprog.elf to the SD card as *autorun.pex* and load code to load it from the SD card.

4.7 –P List available serial ports

Use this option to list all serial ports. Not all of the ports listed will necessarily be connected to boards containing Propeller chips.

4.8 -p <port> Select serial port

Use this option to select the serial port that is connected to the Propeller board you wish to load. If this option is not specified and the environment variable *PROPELLER_LOAD_PORT* is defined, it's value is used. If it is not defined, the loader will search for ports attached to a Propeller board. The first one found is used.

Examples

```
-p COM12
-p /dev/ttyUSB12
-p /dev/cu.usbserial-12
```

If *<port>* begins with a digit under Windows or something other than a '/' under Linux or Mac OS X, it is interpreted as a shorthand for the full port name. In that case, the system-specific prefix is added to *<port>* to form the full name.

Port prefixes

- Windows: “COM”
- Linux: “/dev/ttyUSB”
- Mac OS X: “/dev/cu.usbserial-“

Examples

```
-p12
-p 12
```

Under Windows these would be interpreted as COM12.

Under Linux they would be interpreted as /dev/ttyUSB12.

Under Mac OS X they would be interpreted as /dev/cu.usbserial-12.

4.9 -q Quit on the exit sequence

Use this option to cause propeller-load to exit terminal mode when it receives the byte sequence (0xff, 0x00, status) from the target board. This is primarily intended for use in automated test scripts.

4.10 -r Run the program after loading

Use this option to start the program running after loading has completed.

4.11 –S or –S<n> Slow down the loader by adding a delay

Use this option to introduce a time delay during the initial phase of loading that uses the Propeller boot protocol. You might want to use –S if you find that your computer is too fast for the Propeller and you get load errors. This sometimes happens when you run the loader on a Macintosh. If *<n>* is not given, a delay of 5 microseconds is used. The *<n>* must be immediately adjacent to the –S with no intervening space.

If the computer is too fast, the Propeller can reset in the middle of a download and will boot the program previously saved in EEPROM rather than the one just loaded.

Examples

```
-S
```

This will cause a delay of 5 microseconds to be used.

```
-S12
```

This will cause a delay of 12 microseconds to be used.

4.12 -s Write a spin .binary file for use with the Propeller Tool

Use this option to write a Spin .binary file for use with the Propeller Tool or any other loader that can handle the Spin binary format.

Example

```
-s myprog.elf
```

This will write myprog.binary.

4.13 -t or –t<baud> Enter terminal mode after running the program

Use this option to cause the loader to enter a simple terminal emulator after loading is complete. If *<baud>* is given the baud rate is changed to this value before entering terminal mode. The *<baud>* must be immediately adjacent to the –t with no intervening space.

4.14 -v Verbose output

Use this option to produce more verbose progress information.

4.15 -x Write a .pex binary file for use with the SD loader or SD cache

Use this option to write a Propeller executable file (.pex). This file can then be transferred to an SD card to be run using either the –z or –l option.

Example

```
-x myprog.elf
```

This will read the program myprog.elf and write myprog.pex.

4.16 -z Write a program to the sd card and use the SD cache

Use this option to load code that can run the program *autorun.pex* from the root directory on the SD card. If a filename is given in the command, that file will be written to the SD card as *autorun.pex.* It should be an XMMC program.

Examples

```
-z
```

Load code to run the program *autorun.pex* which should already be on the SD card

```
-z myprog.elf
```

Write the program myprog.elf to the SD card as *autorun.pex* and load code to run it from the SD card.

4.17 -? Display a usage message and exit

Use this option or just invoke propeller-load with no parameters to display a usage message.

5 Configuration Files

The loader uses board configuration files for information specific to each type of target board. The command line option "-b *myboard*" selects the board type *myboard* which causes the loader to read configuration information from the file "myboard.cfg". The loader looks for this file in the include path which is described in section 4.1.

Note: Board configuration files should have lowercase names on Linux since the loader translates the argument to the –b option to lowercase before looking for the corresponding board configuration file.

5.1 Board Types and Subtypes

In it's simplest form, a board configuration file is just a list of variable names and values separated by colons. For example, here is a board configuration file for the Parallax C3 board.

```
clkfreq: 80000000
clkmode: XTAL1+PLL16X
baudrate: 115200
rxpin: 31
txpin: 30
tvpin: 12
cache-driver: c3_cache.dat
cache-size: 8K
cache-param1: 0
cache-param2: 0
sd-driver: sd_driver.dat
sdspi-do: 10
```

```
sdspi-clk: 11
sdspi-di: 9
sdspi-clr: 25
sdspi-inc: 8
sdspi-addr: 5
```

This is contained in a file called “c3.cfg” and can be used by providing the loader command line option “-b c3”.

Sometimes a board will support multiple memory models. One way to handle that is to have a separate configuration file for each memory model but that results in a lot of duplicated information that is common among all of the memory models. A better approach is to use board subtypes. The –b option can accept both a board type and subtype using the syntax “-b *type:subtype*”.

For example, we could have a subtype for the C3 board called “xmmc” that uses a different cache driver. The board configuration file would look like this:

```
clkfreq: 80000000
clkmode: XTAL1+PLL16X
baudrate: 115200
rxpin: 31
txpin: 30
tvpin: 12
sd-driver: sd_driver.dat
sdspi-do: 10
sdspi-clk: 11
sdspi-di: 9
sdspi-clr: 25
sdspi-inc: 8
sdspi-addr: 5

[default]
cache-driver: c3_cache.dat
cache-size: 8K
cache-param1: 0
cache-param2: 0

[xmmc]
cache-driver: c3f_cache.dat
cache-size: 8K
cache-param1: 0
cache-param2: 0
```

Then, if the loader is passed the “-b c3” option, it will use the variables defined in the common section of the configuration file before any of the bracketed tags as well as the variables defined in the section that begins with “[default]”.

However, if the loader is passed the "-b c3:xmmc" option, it will use the variables defined in the common section as well as the variables defined in the "[xmmc]" section.

This subtype feature allows the variables that are common to all memory models to be shared.

5.2 Configuration Variables

Variables known to the loader are:

- clkfreq – clock frequency
- clkmode - clock mode
- baudrate - baudrate
- reset – dtr or rts
- rxpin – receive pin number for second stage load
- txpin – transmit pin number for second stage load
- tvpin – TV pin number for loader debugging
- cache-driver – cache driver filename
- cache-size - cache size in bytes
- cache-param1 – cache driver parameter 1
- cache-param2 – cache driver parameter 2
- sd-driver – SD card driver filename
- sdspi-do – SD SPI DO/MISO pin number
- sdspi-clk – SD SPI clock pin number
- sdspi-di – SD SPI DI/MOSI pin number
- sdspi-cs – SD SPI chip select pin number
- sdspi-clr – SD SPI C3-style clear pin number
- sdspi-inc – SD SPI C3-style increment pin number
- sdspi-start – SD SPI decoder starting bit number
- sdspi-width – SD SPI decoder width in bits
- spdspi-addr – SD SPI decoder or C3-style address
- sdspi-config1 – SD SPI configuration parameter 1
- sdspi-config2 – SD SPI configuration parameter 2
- eeprom-first – TRUE to write loader to EEPROM before writing program

5.3 Expressions

Numeric values in configuration files and passed to propeller-load with the –D option use a subset of C expression syntax. The following operators are allowed:

```
expr ? trueExpr : falseExpr
expr || expr
expr && expr
```

```
expr | expr
expr ^ expr
expr & expr
expr == expr
expr |= expr
expr < expr
expr <= expr
expr >= expr
expr > expr
expr << expr
expr >> expr
expr + expr
expr – expr
expr * expr
expr / expr
expr % expr
+ expr
- expr
~ expr
! expr
( expr )
```

Value expressions can include the following symbols:

```
rcfast
rcslow
xinput
xtal1
xtal2
xtal3
pll1x
pll2x
pll4x
pll8x
pll16x
true
false
```

A number followed by “k” will be multiplied by 1024. For example, “8k” will be interpreted as 8192.

A number followed by “m” will be multiplied by 1024*1024. For example, “2m” will be interpreted as 2*1024*1024= 2097152.

A number followed by “MHz” will be multiplied by 1000000. For example “80MHz” will be interpreted as 80000000.

Configuration variable names can be used in expressions as long as they don’t include embedded hyphens. Configuration variables with embedded hyphens must

be surrounded by braces. For instance, cache-param1 can be used in an expression as {cache-param1}.

6 Standard Driver Configuration

6.1 SPI SD Card Driver

Configure the SPI SD card driver by specifying values for the *sdspi-config1* and *sdspi-config2* variables in the board configuration file. Each of these variables has four byte-wide fields to configure pin numbers and other values.

6.1.1 sdspi-config1

The *sdspi-config1* variable has the following format 0x*iiooccpp* with the following fields:

- *ii* – the DI/MOSI pin number
- *oo* – the DO/MISO pin number
- *cc* – the clock pin number
- *pp* – the protocol byte

6.1.2 sdspi-config2

The protocol byte in *sdspi-config1* is a set of bits. The meaning of the fields in *sdspi-config2* depends on the settings of these bits.

The *sdspi-config2 variable has the format 0xaabbccdd* with the following fields:

- *aa* – cs or clr
- *bb* – inc or start
- *cc* – width
- *dd* - addr

The protocol byte is a bit mask with the following bits defined:

If CS_CLR_PIN_MASK (0x01) is set, then byte *aa* contains the CS or C3-style CLR pin number

If INC_PIN_MASK (0x02) is set, then byte *bb* contains the C3-style INC pin number

If MUX_START_BIT_MASK (0x04) is set, then byte *bb* contains the starting bit number of the decoder field

If MUX_WIDTH_MASK (0x08) is set, then byte *cc* contains the width of the mux field

If ADDR_MASK (0x10) is set, then byte *dd* contains either the C3-style address or the value to write to the mux field

Examples

```
sdspi-config1: (9<<24) | (10<<16) | (11<<8) | 0x13
sdspi-config2: (25<<24) | (8<<16) | 5
```

These values will configure the SD SPI driver for the C3.

```
sdspi-config1: (0<<24) | (1<<16) | (2<<8) | 0x01
sdspi-config1: (3<<24)
```

These values will configure the SD SPI driver for a board where DI/MOSI is on P0, DO/MISO is on P1, CLK is on P2, and CS is on P3.

6.2 Cache Drivers

The cache drivers are configured using the variables *cache-param1* and *cache-param2.*

6.2.1 SPI Flash Cache Driver

The SPI flash cache driver (spi_flash_cache.dat) is configured the same way as the SD driver described above except that *cache-param1* is used in place of *sdspi-config1* and *cache-param2* is used in place of *sdspi-config2.*

6.2.2 SPI SRAM Cache Driver

The SPI SRAM cache driver (spi_sram_cache.dat) is configured the same way as the SD driver described above except that *cache-param1* is used in place of *sdspi-config1* and *cache-param2* is used in place of *sdspi-config2.*

6.2.3 SQI Flash Cache Driver

The SQI flash cache driver (sqi_flash_cache.dat) is configured in a manner similar to the SD driver described above except that *cache-param1* is used in place of *sdspi-config1* and *cache-param2* is used in place of *sdspi-config2. In addition,* the *ii* field in *cache-param1* is the first pin of a four pin group used as D0-D3 on the Quad SPI flash chip and the *oo* field is not used.

6.2.4 EEPROM Cache Driver

The *eeprom_cache.dat* driver only uses *cache-config1* to specify the starting offset in the EEPROM where the program should be loaded. The default setting will be used if the parameter is 0. It is very important to ensure that this offset is beyond the end of the boot program that resides in the EEPROM if the value is not zero.

6.2.5 C3 Cache Driver

c3.dat

This cache driver uses a separate cache for instructions and data. It can be used with the XMM-SPLIT memory model.

The *cache-param1* variable should contain number of bits in the cache line index if it is non-zero. If this variable is missing or if it has the value 0, the default of 6 is used. Call this value *indexWidth.*

The *cache-param2* variable should contain the number of bits in the cache line offset if it is non-zero. If this variable is missing or if it has the value 0, the default of 6 is used. Call this value *offsetWidth.*

Note that the cache-size variable must be equal to:

```
(1 << (indexWidth + offsetWidth + 1))
```

c3f.dat

This cache driver uses a single cache for instructions in the XMMC memory model and a unified cache for instructions and data in the XMM-SINGLE memory model.

The *cache-param1* variable should contain number of bits in the cache line index if it is non-zero. If this variable is missing or if it has the value 0, the default of 6 is used. Call this value *indexWidth.*

The *cache-param2* variable should contain the number of bits in the cache line offset if it is non-zero. If this variable is missing or if it has the value 0, the default of 6 is used. Call this value *offsetWidth.*

Note that the cache-size variable must be equal to:

```
(1 << (indexWidth + offsetWidth))
```

7 Variable Patching

The loader provides a way to automatically configure a program for a specific target board. Often the same program will run on multiple boards by simply changing the pin numbers used to interface with off-chip hardware like TV, VGA, keyboard, etc.

One approach to handling this is to use #defines to configure the program for a particular board but this approach requires that the program be recompiled for each board.

The loader provides a way to do this without recompiling the program. During the load process, the loader looks at the symbol table contained in the program file (.elf file produced by the linker) for symbols whose names begin with "_cfg_". When it finds one of these symbols, it looks for a variable in the selected board configuration file with a matching name and stores the value from the configuration file into the variable before starting the program.

The loader finds matching configuration variable names by first removing the "_cfg_" prefix and then replacing any embedded underscore characters with hyphens. For example, the user variable "_cfg_sdspi_cs" would match the configuration variable "sdspi-cs".

Example

```
int _cfg_baudrate = -1;
```

If the program being loaded contains this variable definition, the loader will replace the value -1 with the value of the "baudrate" variable from the selected board configuration file.

Note that the variable declaration must specify a non-zero initial value for the variable. This is because the linker will place any variable that is not initialized or one that is initialized to zero into a special program area that is zeroed at startup.

There are two exceptions to the rule of finding matching configuration variables. These are the variables "_cfg_sdspi_config1" and "_cfg_sdspi_config2". These variables can either be patched from the corresponding configuration file variables "sdspi-config1" and "sdspi-config2" or the loader will construct their values using the individual configuration file variables for describing the SD card interface: "sdspi-do", "sdspi-di", "sdspi-clk", "sdspi-cs", "sdspi-clr", "sdspi-set", "sdspi-addr", etc.

8 Memory Models

8.1 COG

In the COG memory model (selected by the propeller-elf-gcc option –mcog), all code is compiled to run within the COG but the loader loads the COG image along with code to load it into a COG into hub memory.

8.2 LMM

In the LMM memory model (selected by the propeller-elf-gcc option –mlmm or by default if no memory model option is supplied), code is compiled to run directly from hub memory using an LMM kernel that is loaded into a COG at startup. In this model, it is also possible to include COG images in the form of .ecog files that the linker will place in the address space starting at the address 0xc0000000. The loader will locate those COG images and write them to the boot EEPROM starting at offset 0x8000. This feature will only work on boards that have at least a 64K byte EEPROM.

A .ecog program can be built with the following commands:

```
propeller-elf-gcc -Os -r -mcog –o myprog.ecog myprog.c

propeller-elf-objcopy --localize-text \
         --rename-section .text=myprog.ecog \
         myprog.ecog
```

The linker will provide the following symbols to locate the start and end of the .ecog image.

```
_load_start_myprog_ecog
_load_stop_myprog_ecog
```

You can determine the offset of the .ecog image in EEPROM using the following formula:

```
_load_start_myprog_ecog – 0xc0000000 + 0x8000
```

The size of the .ecog image in bytes is:

```
_load_stop_myprog_ecog - _load_start_myprog_ecog
```

8.3 XMMC

In the XMMC memory model (selected by the propeller-elf-gcc option –mxmmc), code is placed in memory starting at 0x30000000 and all data is placed in hub memory. This memory model is normally used when the board supports flash memory at 0x30000000 such as the Parallax C3. It can also be used if the board supports RAM at that address. Also, some boards do incomplete address decoding and hence appear to have RAM at 0x30000000 even though RAM is normally placed at 0x20000000. In order to use the XMMC memory model with external RAM you must include the following line in the board configuration file:

```
load-target: ram
```

8.4 XMM-SINGLE

In the XMM-SINGLE memory model (selected by the propeller-elf-gcc option –mxmm-single), code and data are placed in memory starting at address 0x20000000. because code and data are intermixed in memory this memory model must be used with external RAM.

8.5 XMM-SPLIT

In the XMM-SPLIT memory model (selected by the propeller-elf-gcc option –mxmm-split or just –mxmm), data is placed in memory starting at 0x20000000 and code is placed in memory starting at 0x30000000. The board must support memory at both locations and the memory at 0x20000000 must be RAM. If the memory at 0x30000000 is also RAM, the following line must be in the board configuration file:

```
load-target: ram
```

Also, the memory at 0x2000000 and 0x30000000 must be independent and not just images of the same memory at both addresses.