

Catalina



Compiler

Reference Manual for the Propeller 1

Table of Content

What is Catalina?	7
Status	7
Features	7
Catalina is ANSI C compliant	8
Catalina runs on Windows, Linux and OSX	8
Catalina supports multiple Propeller platforms	8
Catalina supports C programs up to 16 Mb	11
Catalina is Free!	12
But what does all this really mean?	14
Installing Catalina	15
Overview	15
Catalina Directory Structure	16
Using Catalina	18
Using the Catalina Compiler	18
Catalina Environment Variables	21
Using lcc directly	23
Using the Catalina Binder	23
Using the Payload Loader	25
Interactive Mode	30
Internal terminal emulation	30
External terminal emulation	33
A Note about payload's internal interactive mode under Windows	34
Catalina YModem Support	36
Building the Payload Loader utilities	40
Catalina Support for the Propeller	42
SPIN/PASM Assembler Support	42
Memory Management Support	43
The hub_malloc functions	43
The alloca function	44
Specifying the heap size	45
Floating Point Support	46
HMI Support	47
Keyboard functions	48

<u>Mouse functions.....</u>	<u>49</u>
<u>Screen functions.....</u>	<u>50</u>
<u>Utility functions.....</u>	<u>53</u>
<u>CGI (Computer Graphics Interface) Support.....</u>	<u>53</u>
<u>VGI (Virtual Graphics Interface) Support.....</u>	<u>59</u>
<u>Multi-Processing Support.....</u>	<u>60</u>
<u>Multi-processing and plugins.....</u>	<u>60</u>
<u>Multi-Cog Support.....</u>	<u>60</u>
<u>Multi-Threading Support.....</u>	<u>61</u>
<u>Fundamental Thread Functions.....</u>	<u>62</u>
<u>Additional Thread Utility Functions.....</u>	<u>65</u>
<u>Multi-Model Support.....</u>	<u>66</u>
<u>Producing an array blob:.....</u>	<u>67</u>
<u>Multi-Models and Multi-threading.....</u>	<u>68</u>
<u>Multi-Models and Overlays.....</u>	<u>70</u>
<u>Multi-processing, Locks and Memory Management.....</u>	<u>71</u>
<u>Thread-safe Memory Management.....</u>	<u>71</u>
<u>Memory Fragmentation Management.....</u>	<u>72</u>
<u>Thread, Memory and Service Locks.....</u>	<u>72</u>
<u>Posix threads (pthreads) support.....</u>	<u>74</u>
<u>Posix pthread functions supported.....</u>	<u>74</u>
<u>Posix pthread functions not supported.....</u>	<u>76</u>
<u>Non-Posix pthread functions added.....</u>	<u>77</u>
<u>Random Number Support.....</u>	<u>77</u>
<u>Pseudo Random Numbers.....</u>	<u>77</u>
<u>True Random Numbers.....</u>	<u>78</u>
<u>Random Number Functions.....</u>	<u>78</u>
<u>Plugin Support.....</u>	<u>79</u>
<u>Cog functions.....</u>	<u>79</u>
<u>propeller.h and Special Register Access.....</u>	<u>85</u>
<u>Registry, Plugin and Service functions.....</u>	<u>86</u>
<u>Debugger Support.....</u>	<u>91</u>
<u>BlackBox Support.....</u>	<u>91</u>
<u>POD Support.....</u>	<u>91</u>
<u>Compiling programs for debugging with POD.....</u>	<u>91</u>
<u>Using POD.....</u>	<u>93</u>
<u>SD Card Support.....</u>	<u>94</u>
<u>Real-Time Clock Support.....</u>	<u>95</u>

File System Support.....	95
Serial Device Support.....	100
The tty library (libtty).....	101
The tty256 library (libtty256).....	103
The 4 port Serial library (libserial4).....	104
Sound Support.....	106
SPI/I2C Support.....	108
SPI Flash and Cache Support.....	110
Lua Support.....	111
Embedding Lua scripting in a C program.....	113
Catalina Targets.....	115
Catalina Propeller 1 Targets.....	115
Default Target Configuration Options.....	116
Catalina Hub Resource Usage.....	120
LMM Support.....	121
CMM Support.....	122
XMM Support.....	123
XEPROM Support.....	123
Specifying the Memory Model.....	124
EMM Support.....	129
SMM Support.....	130
Catalina Cog Usage.....	130
Supporting multiple Propeller platforms.....	131
Target Packages.....	132
The standard target package (target).....	132
The embedded target package.....	133
The minimal target package.....	133
Using PASM with Catalina.....	135
Inline PASM.....	136
The PASM function.....	136
The _PASM macro.....	136
The _PSTR macro.....	139
Load the PASM program at initialization time.....	142
Convert the PASM program into a Catalina plugin.....	142
Load a compiled PASM program into a cog.....	142
Writing an LMM PASM function that can be called directly from C.....	143
Precautions when using LMM PASM with the Catalina Optimizer.....	144
Multi-CPU Support.....	145

<u>Proxy Devices.....</u>	<u>145</u>
<u>Generic Proxy Server.....</u>	<u>146</u>
<u>Resetting and/or Loading another Prop.....</u>	<u>147</u>
<u>Catalina XMM SD Loader.....</u>	<u>148</u>
<u>Generic SIO Loader.....</u>	<u>148</u>
<u>CPU_n_Boot.....</u>	<u>149</u>
<u>CPU_n_Reset.....</u>	<u>149</u>
<u>Multi-CPU Examples.....</u>	<u>149</u>
<u>Parallelizer Support.....</u>	<u>150</u>
<u>Customizing Catalina.....</u>	<u>151</u>
<u>Customized Platforms.....</u>	<u>151</u>
<u>Customized Targets and Target Packages.....</u>	<u>152</u>
<u>Using existing Parallax Drivers.....</u>	<u>153</u>
<u>Use a Spin object unmodified.....</u>	<u>153</u>
<u>Use only the PASM portion of the driver.....</u>	<u>154</u>
<u>Building Catalina.....</u>	<u>156</u>
<u>Catalina Technical Notes.....</u>	<u>157</u>
<u>A Note about Binding and Library Management.....</u>	<u>157</u>
<u>A Note about the Catalina Libraries.....</u>	<u>159</u>
<u>A Note about C Program Startup & Memory Management.....</u>	<u>161</u>
<u>A Note about POD and EMM/XMM.....</u>	<u>164</u>
<u>A Note about Catalina Code Sizes.....</u>	<u>165</u>
<u>A Note about Catalina symbols vs C symbols.....</u>	<u>169</u>
<u>A Note about the Catalina Loader Protocol.....</u>	<u>171</u>
<u>Catalina Development.....</u>	<u>173</u>
<u>Reporting Bugs.....</u>	<u>173</u>
<u>If you want to help develop Catalina.....</u>	<u>173</u>
<u>Okay, but why is it called “Catalina”?.....</u>	<u>173</u>
<u>Acknowledgments.....</u>	<u>174</u>
<u>Catalina Internals.....</u>	<u>176</u>
<u>A Description of the LMM and XMM Kernels.....</u>	<u>176</u>
<u>A Description of the Catalina Virtual Machine.....</u>	<u>178</u>
<u>Registers.....</u>	<u>178</u>
<u>Primitives.....</u>	<u>179</u>
<u>Kernel Memory Models.....</u>	<u>183</u>
<u>Unsupported PASM.....</u>	<u>183</u>
<u>Object and Image Formats.....</u>	<u>184</u>
<u>Catalina Calling Conventions.....</u>	<u>185</u>

<u>A Description of the Standard Catalina XMM API.....</u>	<u>187</u>
<u>The XMM API cache access functions.....</u>	<u>187</u>
<u>The XMM API direct access functions.....</u>	<u>188</u>
<u>The XMM API flash access functions.....</u>	<u>190</u>
<u>A Description of the Catalina Addressing Modes.....</u>	<u>192</u>
<u>A Description of the Catalina Propeller 1 Image Format.....</u>	<u>194</u>
<u>A Description of the Propeller 1 Generic SD Loader.....</u>	<u>199</u>
<u>A Description of the Proxy Device Protocol.....</u>	<u>202</u>
<u>SD_Init – enable (initialize) the SD card.....</u>	<u>203</u>
<u>SD_Read – read a sector from the SD card.....</u>	<u>203</u>
<u>SD_Write – write a sector to the SD card.....</u>	<u>203</u>
<u>SD_ByteIO – write a byte to the SD card.....</u>	<u>203</u>
<u>SD_StopIO – disable (tristate) the SD card.....</u>	<u>203</u>
<u>KB_Reset – reset the keyboard (clear any buffered keys).....</u>	<u>204</u>
<u>KB_Data – read a character of keyboard data.....</u>	<u>204</u>
<u>MS_Data – read mouse data.....</u>	<u>204</u>
<u>TV_Data – write screen data.....</u>	<u>204</u>

What is Catalina?

Catalina is a free ANSI C compiler for the Parallax Propeller. It can be downloaded from SourceForge at <http://catalina-c.sourceforge.net/>

Catalina supports both the Propeller 1 (aka **P1** or **P8X32A**) and the Propeller 2 (aka **P2** or **P2X8C4M64P**).

This manual is specifically intended for users of the Propeller 1. Many of the libraries, and several of the supported memory models and plugins are only supported on the P1. For information specific to the Propeller 2, see the ***Catalina Compiler Reference Manual for the Propeller 2***.

Status

For a complete list of enhancements since the last release, see the section later in this document titled **What's new in this release?**

Catalina is fairly light on documentation. There is this document (which contains technical details about Catalina) and also several tutorial documents which describe how to use various parts of Catalina – but all the documents currently assume a fair degree of familiarity with the Propeller, and also some degree of familiarity with the C language. There are also README files in various directories.

However, since Catalina is an ANSI compliant C compiler, most existing documentation on the C language and the standard C libraries is applicable to Catalina – this document therefore concentrates on those aspects of Catalina that are unique, such as its Propeller-specific features.

This means you can begin programming in C *without reading this manual at all* – start with the tutorial guides, such as **Getting Started with Catalina** or **Getting Started with the Catalina Geany IDE** and then come back to this guide to find out more about Catalina.

Features

- ANSI C compliant (C89, with some C99 features);
- Floating point support (32 bit IEEE 754);
- Complete C89 library including file system support (with some C99 functions);
- Full debugger support (source code and/or assembly level debugging);
- Multiple platform support – supports **ANY** Propeller platform;
- Multiple OS support - Win32 and Linux binaries are provided. Catalina also runs on OSX, but it has to be compiled from source;
- Support for C programs larger than 32k;
- Support for a **Geany**-based Integrated Development Environment;
- Support for both the Propeller 1 and the Propeller 2;

- Free, and open source.

Catalina is ANSI C compliant

Catalina is based on the widely used, ANSI compliant “Little C Compiler” (**lcc**). Catalina adds a new code generator back-end to **lcc** specifically to generate code for the Parallax Propeller.

Catalina is C89 compliant, with some C99 features (such as supporting `//` for comments).

A C89 compatible C library is provided. This library is based on the venerable Amsterdam Compiler Kit library. Some C99 compliant components (e.g. **stdint.h** and **sdtype.h**) are included, and various other portable C99 libraries are available if additional C99 support is required¹.

Catalina supports full 32 bit floating point, compliant with both ANSI C and IEEE 754.

For further details on **lcc** see <http://www.cs.princeton.edu/software/lcc/>.

For further details on the Parallax Propeller see <http://www.parallax.com>.

For further details on the Amsterdam Compiler Kit see <http://tack.sourceforge.net/>.

Catalina runs on Windows, Linux and OSX

lcc has been ported to many platforms, and any platform that supports **lcc** can also support Catalina, since the remaining portions of Catalina can themselves be compiled with **lcc**.

Binary releases are supplied for both Windows and Linux platforms. All Catalina source code is supplied, to simplify porting Catalina to other platforms – e.g. Catalina can be built from source to run on OSX.

Catalina supports multiple Propeller platforms

Catalina uses the concepts of *platforms*, *targets* and *target packages* to define the C program execution environment. Each *target package* supports one or more *targets* on one or more Propeller hardware *platforms* (or one or more different configurations of the same platform).

Each *target* defines the memory model and load option to be used for the program, and also initializes the hardware and software environment in which the program is to execute (e.g. to specify that real-time clock support, SD card drivers, or floating point packages need to be loaded).

Each *target* typically supports a set of options that can be specified at compile time to include or exclude various components, or to configure them (e.g. to tell the TV driver whether to use NTSC or PAL mode).

¹ For example, an implementation of the C99 **snprintf** functions is available here: <http://www.ijs.si/software/snprintf/>

The *targets* essentially provide Catalina C programs with a hardware abstraction layer, which means the programs can often be made entirely independent of the environment in which they execute.

On the Propeller 1, Catalina compiles C programs into **LMM** (i.e. Large Memory Mode) files, **CMM** (i.e. Compact Memory Mode, also known as **COMPACT** mode) or **XMM** (i.e. eXternal Memory Mode, also known as **SMALL** or **LARGE**) files which are *not* target-specific. Then the Catalina kernel, the necessary device drivers, and any other platform specific code required for the target are bundled into a single target-specific file, which is also compiled and finally combined with the compiled C program.

Catalina provides several target packages, each in a separate sub-directory²:

<i>target/p1</i>	This is the default Catalina target package for the Propeller 1. It supports many Propeller platforms, all memory models, all load options, and all plugins ³ .
<i>embedded/p1</i>	This is a smaller target package for the Propeller 1. It supports only one Propeller platform (which must be configured by the user) and a limited set of plugins. However, it supports all memory models and all load options.
<i>minimal/p1</i>	This is a very trivial target package for the Propeller 1. It supports only one Propeller platform (which must be configured by the user), one plugin, one memory model and one load option. The purpose of this package is mainly to provide a very simple environment to illustrate how to create new Catalina plugins.

The default target package (i.e. *target/p1*) is flexible enough to accommodate nearly all the possible hardware configurations of all the supported Propeller 1 platforms.

The base **Propeller 1** platforms currently supported are:

- The Parallax **ACTIVITY** board
- The Parallax **DEMO** board
- The Parallax **QUICKSTART** board (including the Human Interface Board add-on)
- The Parallax **FLIP** module
- The **HYDRA**
- The **HYBRID**
- The **TRIBLADEPROP** (all CPUs)
- The **RAMBLADE**

² You may also see sub-directories named with a */p2* suffix, such as **target/p2** – these are specific to the Propeller 2. Refer to the Catalina C Compiler Manual for the Propeller 2 for more details.

³ Plugins are described later in this document. For the present, just think of them like drivers for particular devices – e.g. to communicate with a screen or keyboard.

- The **RAMBLADE3**
- The **DRACBLADE**
- The **ASC** (Arduino **S**hield **C**ompatible)
- The **C3** (Credit **C**ard **C**omputer)
- The **PP** (Propeller Platform)
- **CUSTOM** boards (by default configured for the Parallax **QuickStart** board)

Catalina also supports various XMM “add-on” boards that can be added to any base platform. The add-on boards currently supported are:

- The **HX512**
- The **SUPERQUAD**
- The **RAMPAGE**
- The **RP2** (Ram**P**age 2)
- The Parallax **PMC** (Propeller **M**emory **C**ard)

Each platform or add-on board supported by the package has a corresponding symbol reserved for it (e.g. **HYDRA**) that can be specified on the command line via the **-C** option (command line options are described later in this document). More details on the standard targets and their configuration options are given in the **Catalina Targets** section later in this document.

New symbols can be created for other Propeller-based platforms, or for unusual configurations of the above platforms - see the **HMI Support** and **Customized Targets** sections later in this document.

The **CUSTOM** platform is the default platform, used unless another platform is specified on the command line (more on how to do this later). The **CUSTOM** platform comes preconfigured to be suitable for a Propeller 1 with a 5 Mhz clock, and with serial input and output available. This makes the **CUSTOM** platform suitable for many Propeller boards, including the Parallax QuickStart board and the various Gadget Gangster boards. The **CUSTOM** platform is easily modified if none of the predefined platforms is suitable.

Unless otherwise specified, the remainder of this document assumes that the default Catalina target package (i.e. *target*) and the **CUSTOM** platform are in use.

Catalina supports C programs up to 16 Mb

Catalina uses the **Large Memory Model** (LMM) mode of the Parallax Propeller to support programs up to 32 kb on *any* Propeller 1 platform.

Catalina also introduces a new **Compact Memory Model** (CMM), which can also be used on *any* propeller platform, and typically *halves* code sizes when compared with LMM mode.

It is sometimes said that C – or in fact any LMM-based compiler - generates code sizes too large to be useful on the propeller – but not with Catalina, and especially not with Catalina and CMM. For more details (and an example) of just how useful CMM can be in reducing program code sizes, see the section later in this document called **A Note about Catalina Code Sizes**. This section demonstrates how a C “hello, world” type program can take as few as 125 bytes.

However, no matter how efficient the compiler, sometimes a program is just *too large* to fit in 32 kb – so Catalina also provides **External Memory Model** (XMM) support for programs larger than 32 kb on suitable platforms. XMM support allows Catalina to support program sizes up to 16 Mb on Propellers equipped with suitable hardware.

The main advantage of LMM programs is that they execute many times faster than Spin programs. CMM programs tend to execute more slowly than LMM programs, but are still faster than Spin programs. XMM programs may also execute faster than Spin – it depends on the memory architecture used.

LMM C programs are not executed directly on a “bare metal” Propeller – instead, an LMM *kernel* is first loaded into one or more of the Propeller cogs and these cogs can then execute LMM programs. However, LMM is not “interpreted” in the same way as SPIN – the LMM binary opcodes are true Propeller opcodes – the main difference between an LMM program and a PASM program is that for PASM programs the program code is stored in cog RAM, while for LMM programs the program code is stored in Hub RAM. This means LMM programs are somewhat slower than cog programs - but they are significantly *faster* than SPIN programs, and can be significantly *larger* than pure PASM programs (which are limited to 496 instructions).

CMM programs are executed using a kernel that is a hybrid between an interpreted Spin-type kernel and an LMM kernel.

XMM programs are similar to LMM programs except that the program code is stored in external RAM – i.e. RAM provided by additional hardware external to the Propeller chip. This may be Parallel SRAM or Serial Peripheral Interface (SPI) RAM or Flash.

Catalina provides a Standard Target Package which includes support for CMM and LMM programs on all platforms⁴, and also support for XMM programs on platforms with suitable external memory hardware, or *any* Propeller platform with an EEPROM larger than 32kb.

⁴ Note that there are other implementations of LMM for the Propeller, but they are not compatible with the Catalina LMM Kernel.

Currently, Catalina supports using the HYDRA XTREME HX512 SRAM card⁵ for XMM programs on both the **HYDRA** and the **HYBRID** platforms. Catalina also supports using SRAM installed on the **RAMBLADE**, the **RAMBLADE3**, the **TRIBLADEPROP** (both on CPU #1 and CPU #2), or the **DRACBLADE** for XMM programs. Finally, Catalina also supports using SRAM and FLASH on the **HX512**, **C3**, **SUPERQUAD**, **RAMPAGE**, **RAMPAGE2** and **PMC** XMM add-on boards for XMM programs. Other XMM hardware may be supported in future releases.

Catalina is Free!

Catalina is derived from various sources, and so various license conditions apply to different parts of it. However, all components are essentially “free” in that they can be used for any purpose, modified in any way, and re-released - provided such releases comply with the appropriate license conditions.

For example, some parts of Catalina incorporate (or are derived from) Parallax software (e.g. the Catalina Human Machine Interface device drivers) and are distributed under the MIT license (for details, see the individual source files).

lcc itself (apart from the Catalina Code Generator) is covered by a separate “fair use” license. See the file **CPYRIGHT** in the directory **source\lcc** included in each source distribution of Catalina. One of the terms of that license is that developers of products that use **lcc** must request that all bug reports on their product be reported to them – so see the **Reporting Bugs** section later in this document.

The Catalina Target Package (i.e. the components of Catalina that end up incorporated into applications compiled with Catalina, such as the kernel) is distributed under the terms of the GNU Lesser General Public License (LGPL), plus the following special exceptions:

- Use of the Catalina Binder (or any other tool) to combine application components with Catalina Target Package (CTP) components does not constitute a derivative work and does not require the author to provide source code for the application, or provide the ability for users to link their applications against a user-supplied version of the CTP.

However, if you link the application to a *modified* version of the CTP, then the changes to the CTP must be provided under the terms of the LGPL in sections 1, 2, and 4.

- You do not have to provide a copy of the CTP license with applications that incorporate the CTP, nor do you have to identify the CTP license in your program or documentation as required by section 6 of the LGPL. However, applications must still identify their use of the CTP. The following example statement can be included in user documentation to satisfy this requirement:

⁵ To use the full 512 kb of RAM available on the HX512 SRAM card requires the installation of Eric Moyer's firmware modifications. A copy of the firmware is included in the Catalina utilities directory, and is also available at <http://forums.parallax.com/forums/default.aspx?f=33&m=196587>

[application] incorporates components provided as part of the Catalina C Compiler for the Parallax Propeller.

Each of the affected CTP components contains the following license details:

```

-----
'
'   Copyright 2009 Ross Higson
'
'   The portion of this file identified as the LMM Kernel is part of the
'   Catalina Target Package.
'
'   The Catalina Target Package is free software: you can redistribute
'   it and/or modify it under the terms of the GNU Lesser General Public
'   License as published by the Free Software Foundation, either version
'   3 of the License, or (at your option) any later version.
'
'   The Catalina Target Package is distributed in the hope that it will
'   be useful, but WITHOUT ANY WARRANTY; without even the implied warranty
'   of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
'   See the GNU Lesser General Public License for more details.
'
'   You should have received a copy of the GNU Lesser General Public
'   License along with the Catalina Target Package.  If not, see
'   <http://www.gnu.org/licenses/>.
'
-----

```

The exceptions are stated in the README file included in each CTP. A full copy of the LGPL is in the file called COPYING.LESSER, included with each of the target packages distributed with Catalina.

The other significant parts of Catalina – i.e. the Catalina Code Generator and the Catalina Binder - are distributed under the terms of the GNU General Public License (GPL).

Each of these components contains the following license details:

```

-----
'
'   Copyright 2009 Ross Higson
'
'   This file is part of Catalina.
'
'   Catalina is free software: you can redistribute it and/or modify
'   it under the terms of the GNU General Public License as published by
'   the Free Software Foundation, either version 3 of the License, or
'   (at your option) any later version.
'
'   Catalina is distributed in the hope that it will be useful,
'   but WITHOUT ANY WARRANTY; without even the implied warranty of
'   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
'   GNU General Public License for more details.
'
'   You should have received a copy of the GNU General Public License
'   along with Catalina.  If not, see <http://www.gnu.org/licenses/>.
'
-----

```

A full copy of the GPL is included with all distributions, in the file called COPYING.

For more information about the GPL or the LGPL, refer to that file or visit <http://www.gnu.org/licenses>.

But what does all this *really* mean?

All that stuff in the previous section basically means you can use Catalina - for any purpose - ***completely free of charge***.

It also means that Catalina can be used to create *commercial applications for which you can charge*, without those applications having to be released under the GPL. Acknowledging the use of the Catalina Target Package is usually as much as you will need to do.

However, anyone intending to create such an application should read the previous section in detail, and (particularly if you use the C89 library), you should also check the licenses of each component of that library to make sure they are compatible with the license under which your application is to be released.

Of course, an application that incorporates (in whole or part, modified or unmodified) those parts of Catalina which are covered by the GPL (such as the Catalina Binder or the Catalina Code Generator) must still itself be released under the GPL. This just means you can't take Catalina – either in whole or part – and sell it (or a derivative of it) as your own work.

Installing Catalina

Overview

On Windows, Catalina comes with a “one touch” installer that can install both the sources and binaries. It can also install both a command line shortcut to allow easy use of **Catalina** from the command line, and a version of **Geany** for those who prefer to use an integrated development environment (IDE).

If you are using this installer, simply follow the instructions in the installer itself – however, it is recommended that you read this section anyway, as it contains useful information (e.g. on the directory structure that will be installed).

Note that the Catalina installer will install version 3.4.9 of the Cygwin DLL (*cygwin1.dll* in the Catalina *bin* directory). This is licensed under the GNU Lesser GPL version 3. A copy of this license is included in the file *COPYING.LESSER*.

However, if you intend to also use other Cygwin-based software, it is recommended by Cygwin that you instead install the Cygwin DLL as part of the Cygwin distribution (see www.cygwin.org). In that case, simply make sure that version of the Cygwin DLL is on your PATH, and delete the file *bin\cygwin1.dll*

On Linux, you can use **gzip/tar**⁶ to extract the entire distribution into the folder in which Catalina is to be installed – the standard location is */opt/catalina*. However, if you need to rebuild Catalina to suit your Linux distribution, you should first install it elsewhere, rebuild it, and then copy it to this location. See the file *BUILD.TXT* for more details.

Installing to a directory other than the standard location is possible, but it means that some additional setup or options will need to be specified when using Catalina.

⁶ Note that when using **tar**, the **-p** tar option should be specified to preserve file permissions.

Catalina Directory Structure

Wherever Catalina is installed, the directory structure should be something like:

```

Catalina
|
+--- bin
|
+--- catalina_geany
|
+--- demos
|   |
|   +--- benchmarks
|   +--- catalyst
|       |
|       +--- catalina
|       +--- core
|       +--- demo
|       +--- dumbo_basic
|       +--- fymodem
|       +--- image
|       +--- jzip
|       +--- lua_x.y.z
|       +--- pascal
|       +--- sst
|       +--- xvi_x.y
|
|   +--- debug
|   +--- multicore
|   +--- multithread
|   +--- multimodel
|   +--- spinc
|   +--- minimal
|   +--- serial4
|   +--- (etc)
|
+--- documents
|
+--- embedded
|   |
|   +--- p1
|
+--- include
|   |
|   +--- sys
|
+--- lib
|   |
|   +--- p1
|       |
|       +--- cmm
|           |
|           +--- c
|           +--- ci
|           +--- cx
|           +--- cix
|           +--- m
|           +--- ma
|           +--- mb
|           +--- mc
|           +--- (etc)
|       |
|       +--- lmm
|           |
|           +--- (as above)
|       |
|       +--- xmm
|           |
|           +--- (as above)
|   |
|   +--- p2
|       |
|       +--- cmm
|           |
|           +--- (as above)
|       |
|       +--- lmm
|           |
|           +--- (as above)

```



```

|
|       +--- lmm
|       |
|       +--- (as above)
|
|       +--- nmm
|       |
|       +--- (as above)
|
|       +--- xmm
|       |
|       +--- (as above)
|
+--- minimal
|
|       +--- p1
|
+--- source
|
|       +--- catalina
|       +--- catoptimize
|       +--- comms
|       +--- lcc
|       +--- lib
|       +--- openspin
|       +--- p2asm_src
|
+--- target
|
|       +--- p1
|       |
|       +--- p2
|
+--- utilities
|
+--- validation

```

There may be more or less sub-directories to those shown, depending on which parts of Catalina have been installed – but the *bin*, *include*, *lib*, & *target* directories are the minimum required to use Catalina on any platform.

In this document, which is specific to the Propeller 1, the various **p2** sub-directories will not be discussed in detail. Refer to the **Catalina C Reference Manual for the Propeller 2** for details on those sub-directories.

The path to the main Catalina directory must be added to the appropriate environment variable (e.g. by modifying the **PATH** environment variable).

A batch script to do this (`use_catalina`) is provided in the main Catalina directory. Unless you modify your **PATH** variable to include the Catalina **bin** directory (or use the Catalina Command Line shortcut) this command should be executed each time a command shell is started.

Under Windows the command to use is **use catalina**.

Under Linux the command to use is **source use catalina**.

NOTE: You do not need to *rebuild* Catalina just to *use* it, even if you install Catalina to a location other than the default – but if you ever do need to rebuild it then you may need to modify various sources, make files and batch files – do a search for the term “Program Files (x86)” and replace it appropriately.

Using Catalina

Even though Catalina supports a version of the **Geany** Integrated Development Environments (IDEs), Catalina – like most compilers – is still primarily a command line compiler, and it is recommended that you become at least slightly familiar with using Catalina from the command line even if you intend to mostly use it from **Geany**.

This section contains a brief introduction to using Catalina – for a fuller tutorial-style introduction to Catalina, see the document **Getting Started with Catalina**. This tutorial concentrates on the command line use of Catalina. A tutorial on using Geany with Catalina is also provided, called **Getting Started with the Catalina Geany IDE**.

Using the Catalina Compiler

The Catalina Compiler is invoked using the command `catalina` from within a command shell – this command is a front end for the Little C Compiler (**lcc**), the Spin Compiler (**spinnaker**), and the Catalina Binder (**catbind**) – under most circumstances those programs don't need to be invoked separately.

If you have installed Catalina to a non-standard location (*C:\Program Files (x86)\Catalina* under Windows, or */opt/catalina* under Linux) then you will need to set the **LCCDIR** environment variable to that location (this is described in more detail in the section titled **Catalina Environment Variables**).

For example, under Windows you would say:

```
set LCCDIR=<path to Catalina>
```

NOTE: Do not use quotation marks in the Windows **set** command.

Under Linux (if using the bash shell) you would say:

```
LCCDIR=<path to Catalina>; export LCCDIR
```

Then you can execute the `use_catalina` batch file, which will do the rest of the setup. Under Windows the command to use is:

```
use_catalina
```

Under Linux (if using the bash shell) the command to use is:

```
source use_catalina
```

Assuming this batch file has been executed, or another mechanism has been used to include the Catalina bin directory in the current path, a C program can be compiled using a command similar to:

```
catalina [files | options] ...
```

For example:

```
catalina hello_world.c -lc
```

By default, Catalina compiles each C file specified on the command line to an LMM PASM file, then includes additional files for any required library functions, combines the results into a single file and then invokes **spinnaker** to assemble this file and

produce a binary file. Catalina then combines this compiled program with the necessary target files to produce the final binary executable.

The following list describes the command line options supported by the Catalina Compiler for the Propeller 1 (Propeller 2 specific options are not included):

<code>-? or -h</code>	print this help (and exit)
<code>-b</code>	generate a binary output file (this is the default)
<code>-c</code>	compile only (do not bind)
<code>-d</code>	output diagnostic messages
<code>-C symbol</code>	define Catalina symbol (e.g. <code>-C HYDRA</code>)
<code>-D symbol</code>	define symbol (e.g. <code>-D printf=tiny_printf</code>)
<code>-e</code>	generate an eeprom output file
<code>-g[level]</code>	generate debugging information (default level = 1) ⁷
<code>-H addr</code>	address of top of heap
<code>-I path</code>	include file path (e.g. <code>C:\Program Files (x86)\Catalina\include</code>)
<code>-l lib</code>	search library lib when binding
<code>-k</code>	kill (suppress) the output of compilation statistics
<code>-L pat</code>	path to libraries (e.g. <code>C:\Program Files (x86)\Catalina\lib</code>)
<code>-M size</code>	maximum memory size (use with <code>-x</code>)
<code>-o name</code>	name of output file (default is first file name)
<code>-O[level]</code>	optimize code (default level = 1) ⁸
<code>-p ver</code>	Propeller Hardware Version (ver = 1 or 2)
<code>-P addr</code>	address for Read-Write segments
<code>-R addr</code>	address for Read-Only segments
<code>-R size</code>	size of Read/Write segments
<code>-s</code>	compile to assembly code (do not bind)
<code>-t name</code>	name of dedicated target to use
<code>-T path</code>	target file path (e.g. <code>C:\Program Files (x86)\Catalina\target</code>)
<code>-U symbol</code>	undefine symbol (e.g. <code>-U DEFAULT</code>)

⁷ When using the `-g` option a space cannot be included between the option and the parameter. For example `-g` is valid, and `-g3` is valid – but `-g 3` is not valid. See the **BlackBox Reference Manual** for more information on using `-g` and `-g3`

⁸ When using the `-O` option a space cannot be included between the option and the parameter. For example `-O` is valid, and `-O2` is valid – but `-O 2` is not valid. The Catalina Code Optimizer is not included with the free version of Catalina. If you have purchased it separately, refer to the **Catalina Optimizer Reference Manual** for details.

<code>-v</code>	verbose (output information messages)
<code>-v -v</code>	very verbose (more information messages)
<code>-W option</code>	option to pass directly to lcc
<code>-x layout</code>	use specified segment layout (layout = 0 .. 6, 8 .. 10)
<code>-y</code>	generate listing file

Anything not recognized as a valid option is passed directly to **lcc**. Typically, these are the names of one or more C files to be compiled – but they may also be **lcc** options.

The exit code from the command is zero on a successful compile, non-zero on error.

As an example, a Catalina command to link with the standard C library, and generate a listing might look like:

```
catalina hello_world.c -lc -y
```

More examples are given in the document **Getting Started with Catalina**.

In addition to the options described above, Catalina allows for customization of the target package on the command line by allowing the definition of Catalina symbols. A complete list of symbols recognized by the default target package is given in the section titled **Default Target Configuration Options**. Catalina symbols are defined using the **-C** option. There is generally a specific Catalina symbol defined for each platform and/or significant platform configuration option. For example, to select the Credit Card Computer platform and the high-resolution VGA driver from the target package, you might use a command like:

```
catalina -p2 hello_world.c -lc -C C3 -C HIRES_VGA
```

For more details on support for particular Propeller 1 platforms check if there is a file called **platform_README.TXT** in the *target* folder. For example:

C3_README.TXT

ACTIVITY_README.TXT

DEMO_README.TXT

QUICKSTART_README.TXT

If Catalina is not installed into the expected directory, then additional command line options or environment variables can be used to tell Catalina where to find various files and programs it needs.

NOTE: A common problem is to use the incorrect case for options. Case is significant for all options to the Catalina Compiler, so **-t** is not the same as **-T**.

Catalina accepts long file names, but where used on the command line any file or path names that contain spaces need to be quoted. For example:

```
catalina "C:\Program Files (x86)\Catalina\demos\hello_world.c" -lc -y
```

Catalina Environment Variables

The Catalina Compiler can also use the following environment variables:

<code>CATALINA_DEFINE</code>	a list of symbols to define before the compilation
<code>CATALINA_INCLUDE</code>	a list of paths to search for include files
<code>CATALINA_LIBRARY</code>	the directory where libraries are located
<code>CATALINA_TARGET</code>	the directory where target files are located
<code>CATALINA_TEMPDIR</code>	the directory lcc will use for temporary files
<code>CATALINA_LCCOPT</code>	any options to be passed directly to lcc
<code>LCCDIR</code>	the directory various programs (not just lcc) expect to find other files needed during compilation

These variables are set using normal Windows or Linux commands. For example in Windows, environment variables can be set using a command like:

```
set LCCDIR=C:\Program Files (x86)\my_catalina
```

and cleared using a command like:

```
set LCCDIR=
```

or

```
unset LCCDIR
```

In Linux, the appropriate commands (if using the bash shell) to set an environment variable would be a command like:

```
LCCDIR=/usr/me/my_catalina; export LCCDIR
```

and to clear it would be a command like:

```
unset LCCDIR
```

Catalina also provides a convenient command (`catalina_env`) to display the current value of the above environment variables.

In environment variables, path names do not usually need to be quoted even if they contain spaces.

The `CATALINA_DEFINE` environment variable can be used to specify a list of symbols that will be defined before invoking the compiler and/or binder. Multiple symbols can be separated by a space, comma, semicolon or colon. The main purpose of this is to define symbols that tell the target about the platform on which the programs are to be run – this allows the target to correctly select the platform-specific features (such as the pin definitions to use) and also the appropriate plugins and drivers to load. For example, this variable might be set to `TRIBLADEPROP:CPU_1` OR to `HYDRA` OR `HYBRID`

The `CATALINA_INCLUDE` environment variable can be used to specify where the compiler should look for include files. This may be a list of paths - on cygwin or linux the entries in this list must be separated by a colon (':') while on Windows they must be separated by a semicolon (';'). Any include paths specified on the command line

(i.e. via the `-I` option) are added to the beginning of this list (which means they will be searched first - this can be used to effectively override any default paths, or paths set using the environment variable).

The `CATALINA_LIBRARY` environment variable tells the compiler where to look for libraries. This variable should contain a single path or directory name. Note that Catalina always looks in two locations for libraries - in the current directory, and in the specified library directory. If the `-L` option is specified on the command line it will override this environment variable. The compiler will automatically add `\lib` to the library path for **TINY** and **SMALL** programs, `\compact_lib` for **COMPACT** programs, and `\large_lib` for **LARGE** programs.

The `CATALINA_TARGET` environment variable tells the compiler where to look for the target package. It should contain a single path or directory name. If the `-T` option is specified on the command line it will override this environment variable.

The `CATALINA_TEMPDIR` environment variable tells all programs where to create any temporary files needed during compilation. It should contain a single path or directory name.

The `LCCDIR` environment variable tells all programs (not just **lcc**!) where to find files which are needed during the compilation process. It should contain a single path or directory (e.g. *C:\Program Files (x86)\Catalina*) which has at least the following sub-directories:

bin	sub-directory for executable files
lib\p1\lmm	default sub-directory for library files when using TINY mode or XMM SMALL mode.
lib\p1\cmm	default sub-directory for library files when using COMPACT mode
lib\p1\xmm	default sub-directory for library files when using XMM LARGE mode
include	default sub-directory for include files
target\p1	default sub-directory for target files

Note that the default library and target paths can be overridden by the `CATALINA_LIBRARY` and `CATALINA_TARGET` environment variables. The default include path can effectively be overridden by the `CATALINA_INCLUDE` environment variable since any paths specified there are searched before the default include path.

The `CATALINA_LCCOPT` environment variable can be used to specify options to be passed straight to **lcc**. The entire contents of this environment variable are simply added to the **lcc** command - *before* any options generated by the Catalina command. Remember that the options must be **lcc** options (i.e. they are neither **catalina** options nor **catbind** options) - but also remember that it is possible to specify binder options to **lcc** by prefixing them with `-WI` - i.e. the **lcc** option `-WI-XXX` actually gets passed to the binder as option `-XXX`.

NOTE: Use the `CATALINA_LCCOPT` feature with care. It is possible to specify **lcc** options which will cause Catalina to generate incorrect code, or have other unexpected results.

If in doubt about what **lcc** options are in effect, use the `-v` option to **catalina** to print out the actual **lcc** command that will be executed.

Using lcc directly

Normally, **lcc** is invoked automatically as required by Catalina – but **lcc** can also be called directly. **lcc** itself is quite well documented elsewhere – e.g. see the **lcc** Unix man page located at <http://www.cs.princeton.edu/software/lcc/doc/lcc.1.html>.

Also remember that you can use the `-v` flag to Catalina to see what options Catalina itself uses when invoking **lcc**.

Note that the version of **lcc** provided with Catalina is intended specifically for use as part of the Catalina Propeller cross-compiler. If you need a version of **lcc** for compiling native C programs you should download the original **lcc** sources from <http://www.cs.princeton.edu/software/lcc/> and compile them yourself (make sure to use a separate directory to the one used by Catalina).

Windows users could also try **lcc-win32** (<http://www.cs.virginia.edu/~lcc-win32>).

Using the Catalina Binder

Normally, the Catalina Binder (**catbind**) is invoked automatically as required by **catalina**. However, there are occasions when it may be useful to use the binder separately. For example:

- To bind a Catalina LMM PASM program (e.g. if **catalina** was used with the `-S` option, or to rebind a previously compiled program to a new target). For example, to bind the LMM PASM file **file.s** with the math library and then generate an eeprom image containing the result named **test**, use the following command:

```
catbind file.s -lm -e -o test
```

- To index a set of library files (which are just LMM PASM files that have been compiled using Catalina but not yet bound). For example, to index all symbols imported and exported by all LMM PASM files in the current directory, and then put the result in a file called **catalina.idx** use the following command:

```
catbind -i -e *.s -o catalina.idx
```

More examples are given in the document **Getting Started with Catalina**.

The following list describes all the command line options supported by the Catalina Binder:

- | | |
|------------------------------------|---|
| <code>-?</code> or <code>-h</code> | print this helpful message (and exit) |
| <code>-a</code> | no assembly (output bound source files only) |
| <code>-d</code> | output diagnostic messages (<code>-d -d</code> for even more messages) |

<code>-C symbol</code>	#define 'symbol' before assembling the code
<code>-e</code>	generate export list from input files
<code>-f</code>	force (continue even if errors occur)
<code>-g[level]</code>	generate debugging information (default level = 1) ⁹
<code>-H addr</code>	address of top of heap
<code>-i</code>	generate import list from input files
<code>-k</code>	kill (suppress) the output of compilation statistics
<code>-L path</code>	path to libraries (default is 'C:\Program Files (x86)\Catalina\lib\')
<code>-l name</code>	search library named 'libname' when binding
<code>-M size</code>	memory size to use (used with <code>-x</code> , default is 16M)
<code>-o name</code>	output results (generate, bind or assemble) to file 'name'
<code>-O[level]</code>	optimize code (default level = 1) ¹⁰
<code>-p ver</code>	Propeller Hardware Version (ver = 1 or 2)
<code>-R size</code>	size of Read/Write segments
<code>-T path</code>	target file path (default 'C:\Program Files (x86)\Catalina\target')
<code>-t name</code>	use target 'name'
<code>-u</code>	untidy mode – do not delete intermediate files
<code>-U symbol</code>	do not #define 'symbol' before assembling the code
<code>-v</code>	verbose (output information messages)
<code>-v -v</code>	very verbose (more information messages)
<code>-w opt</code>	pass option 'opt' to the assembler (e.g. <code>-w-l</code> , <code>-w-b</code> , <code>-w-e</code>)
<code>-x layout</code>	use specified segment layout (layout = 0 .. 6, 8 .. 10)
<code>-z ch</code>	specify separator char for path names (default is '\')

The exit code from the command is the number of undefined/redefined symbols (-1 for other errors).

NOTE: A common problem is to use the incorrect case for options. Case is significant for options to the Catalina Binder, so `-t` is not the same as `-T`.

⁹ When using the `-g` option a space cannot be included between the option and the parameter. For example `-g` is valid, and `-g3` is valid – but `-g 3` is not valid. See the **BlackBox Reference Manual** for more information on using `-g` and `-g3`

¹⁰ When using the `-O` option a space cannot be included between the option and the parameter. For example `-O` is valid, and `-O2` is valid – but `-O 2` is not valid. Refer to the **Catalina Optimizer Reference Manual** for details.

Using the Payload Loader

Catalina provides a loader program (called **payload**) that can be used to load Catalina binaries into the Propeller from a PC. The Catalina payload loader is somewhat similar to the Parallax *propellant* program, but with the following Catalina-specific features:

- It runs under both Linux and Windows;
- It can load Catalina XMM binaries¹¹.
- It includes a built-in terminal emulator.
- It can load multiple files in succession, through multiple ports.
- It can load programs on platforms where the normal serial port (on pins 30 & 31) cannot be used when XMM is installed (such as the **HYBRID** or **HYDRA**).
- It can auto-detect both Propeller 1 and Propeller 2 chips.

At its simplest, payload is quite trivial to use. For example, to load **program.binary**, the payload command might be as follows:

```
payload program
```

The above command will cause payload to search all the available serial ports for the first one with a Propeller attached (by default it starts at port 1 and tries each consecutive port in turn) and then load the specified program binary using the first such port it finds at the default baud rate.

Note: if a **.binary** or **.eeprom** extension is not specified, **.bin** is tried first, then **.binary**. This becomes important if you sometimes compile for both the Propeller 1 and the Propeller 2 – if in doubt, specify the extension in full.

The default baud rate used for loading **.bin** files is 230,400 baud, suitable for a Propeller 2. The default baud rate for loading other files (e.g. **.binary**) is 115,200 baud, suitable for a Propeller 1.

Note: under Linux it is important that the user using the loader has read/write access to the port to be used – otherwise the loader will be unable to open the port and will probably report that no propeller is connected. To give user <username> permanent access to the serial ports, the user needs to be added to the 'dialout' user group. This can be done using the following command:

```
sudo usermod -a -G dialout <username>
```

Note: you will have to log out and log back in for this command to take effect.

To find out the name of the last USB serial device just plugged in under Linux:

```
dmesg | grep tty
```

¹¹ Catalina **LMM** binaries are indistinguishable from normal Propeller binaries, and can be loaded using any loader program. But Catalina **XMM** binaries can only be loaded using a Catalina specific loader

Payload commands can get more complex, because payload can be used to load multiple files in succession. To see why this is desirable, first consider the Propeller built-in loader capability – i.e. after reset, the Propeller will respond to a program being loaded via a serial port on pins 30 and 31. However, this built-in loader can only be used to load programs into Hub RAM or EEPROM – it knows nothing about any external XMM SRAM or FLASH that may be connected to the Propeller. To load a program into XMM memory, payload must first load *another* loader – one that knows how to use the XMM memory. A payload command to do this might look as follows:

```
payload XMM my_xmm_program
```

The above command first loads *XMM.binary*, which is itself a loader that knows how to load other files. This first binary is loaded using the built-in Propeller loader. When this program is started, it expects a second file (in this case *my_xmm_program.binary*) to be loaded using a Catalina-specific protocol and it loads that program into XMM memory (and then starts it executing). Payload handles both protocols seamlessly – in this case using the same serial port for both loads.

Even more complex payload commands may be required when the same port cannot be used to do both loads – e.g. when loading programs on platforms such as the **HYDRA** or **HYBRID** (this is because the XMM RAM cannot be used at the same time as the normal serial port).

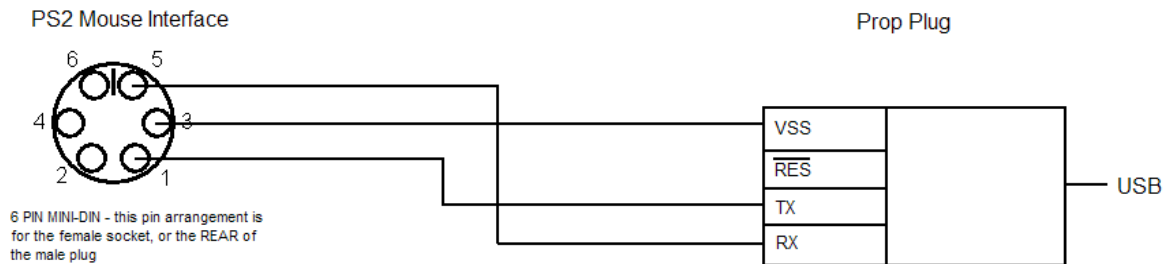
In such cases, the two-step load process is required, and the second load must also use a different port. For example, to load an XMM program on a **HYBRID**, a command similar to the following might be used:

```
payload -p 2 Hybrid_Mouse -s 9 my_xmm_program
```

The above command will *first* load the *Hybrid_Mouse.binary* program into the Propeller using port 2, and *second* will load the program *my_xmm_program.binary* into the Propeller using port 9.

If the ports are not specified, Payload will attempt to find them automatically. However, this may take a long time, and if you have multiple propellers connected this process may find the wrong propeller. Also, some serial or USB devices may cause the auto-detection process to fail. In such cases the ports should be specified manually. This may also speed up the load process, since the auto-detection process must interrogate each port in turn until it finds the one that responds correctly.

The *Hybrid_Mouse.binary* program used in the example above is similar to the *XMM.binary* program used previously, except it expects to load a program using the Propeller pins normally used for communicating with a serial mouse. To use this loader requires a simple cable to be constructed that plugs into the mouse port on one end, and into a Prop Plug on the other, as follows:



This cable can be used in either a mouse or keyboard port, and is used *in addition* to the normal serial connection to the Propeller via pins 30 and 31. This cable can be used to load *any* program, but is specifically intended to be used to load *XMM* programs, once a suitable loader (i.e. the Mouse Loader) has been loaded via the normal serial connection. After the XMM program has been loaded, the normal keyboard or mouse can be replaced for normal program use.

One generic and one specific loader are provided in the *utilities* directory for use with the payload program:

Payload_Loader.spin

This program can be used on any platform that allows the use of the normal serial port to load XMM platforms (e.g. the **DracBlade**, **RamBlade**, **Ramblade3**, **TriBladeProp**, **C3**, **SuperQuad**, **RamPage**, **RamPage2** or **Propeller Memory Card**). It is compiled automatically by the **build_all** batch file in the utilities directory, and named as *XMM.binary*. If the compiled binary is specific to a CPU in a multi-CPU system, it is named *XMM_n.binary*, where **n** is the CPU number. This loader expects the second file to be loaded through the normal serial port, and does not require the use of a special cable.

Mouse_Loader.spin

This program can be used on the Hydra and Hybrid to load XMM programs. It is compiled automatically by the **build_all** batch file in the utilities directory, and named as either *Hydra_Mouse.binary* or *Hybrid_Mouse.binary*. Note that these two platforms use different pins for the mouse port, and therefore the binaries are different – the program uses the pin definitions specified in the file *Catalina_Common.spin*, and could also be compiled for other platforms if required. A keyboard version could also be created if required. This loader expects the second file to be loaded using the mouse port, and requires the use of a special cable like the one shown above.

The loader program binary must always be specified as the first program to be loaded. The normal LMM or XMM program to be loaded is then specified as the second file.

Note that on multi-CPU platforms it may be desirable to do a *three* step load in order to load multiple CPUs – e.g. the initial file might be a boot loader loaded into CPU 1 that (in turn) loads an embedded boot loader into CPU 2, and then configures itself to act as a relay. The second file would be a program that CPU 1 simply relays to CPU 2. After that load is complete, the CPU 1 loader reconfigures itself to accept a third file which it loads into its own CPU. Payload can support this kind of multi-step load process, although a suitable intermediate “relay” loader is not provided for any platform.

Compiling the utilities for a platform can be done using the interactive batch file **build_utilities**. This batch file accepts no parameters – it prompts for all required information (including the platform), and then compiles all the utilities appropriate for the specified platform. See the section **Building the Payload Loader utilities** (below).

Once the utilities directory has been compiled, the loaders can be copied to any working directory for use with the payload loader, or into the Catalina bin directory to save having to specify the path to the utility each time (this is done automatically by the **build_utilities** script). This means that to load an XMM program, the command can be as simple as:

```
payload XMM program
```

If it is necessary to use a particular loader (e.g. if you work with multiple Propeller platforms), a script or batch file can be created that specifies the loader to use. For example, to use the particular mouse port XMM loader designed for the Hydra, a file called *hydraload.bat* might be created to contain the line:

```
payload Hydra_Mouse.binary $1 $2 $3 $4 $5
```

If the same port is always used, this parameter could also be included in the command. Example scripts to simplify loading of XMM programs are provided in the *utilities* directory (e.g. **xmm_payload**) – modify them to suit your needs, and then copy them to the *bin* directory – then you can load XMM programs as easily as non-XMM programs.

The following list shows the options supported by Catalina Payload:

- | | |
|----------|---|
| -? or -h | print a help message and exit (-v -h prints more help, such as a list of supported serial port numbers) |
| -a port | find the ports to use automatically, starting at the specified port (the default if no -a option is specified is to start at port 1) |
| -A key | set attention key (default is 1, 0 disables) |
| -b baud | use the specified baud rate (the default is 115200 when loading .binary or .eeprom files, or 230400 when loading .bin files) |
| -c cpu | cpu destination for the catalina upload (default is 1) |
| -d | diagnostic mode (-d again for more diagnostics) |

-e	program the EEPROM with the loaded program and then start it (otherwise the program is just loaded into RAM and started).
-i	start the internal interactive terminal emulator once the program is loaded, or immediately if no program is to be loaded.
-I term	start the external interactive terminal emulator term once the program is loaded, or immediately if no program is to be loaded.
-j	disable lfsr check altogether
-f msec	set interfile delay in milliseconds (default is 100)
-g col,row	set the number of columns and rows to use in interactive mode
-m max	set maximum retry attempts (default is 5)
-n msec	set sync timeout in milliseconds (default is 100)
-o vers	override Propeller version detection (vers 1 = P1, 2 = P2)
-p port	use the specified port for uploads (or just the first upload if -s is also specified)
-q mode	line mode (1=ignore CR, 2=ignore LF, 4=CR to LF, 8=LF to CR 16=Auto CR on LF Output,32=CR or LF moves cursor NOTE: modes can be combined, e.g. -q3 = -q1 -q2.
-r msec	set reset delay in milliseconds (default is 0)
-s port	switch to the specified port for the second and subsequent uploads
-S msec	set YModem char delay time in milliseconds (default is 0)
-t msec	set read timeout in milliseconds (default is 250)
-T msec	set YModem timeout in milliseconds (default is 3000)
-v	verbose mode (also includes port numbers in the help message)
-w	wait for a key press between each load – useful if you only have one Prop Plug and need to swap it to the mouse port cable before proceeding with the second load
-x	do catalina upload only (i.e. assume the boot loader has already been loaded – e.g. it may be permanently loaded into EEPROM)
-y	do not display download progress messages
-z	do two resets before the initial load (may be required on some platforms)

The **-A** option is for configuring the attention key, that pops up a menu that can be used to select a Terminal Configuration dialog (which can be used to configure the terminal line mode), a YModem file transfer dialog, or to terminate payload. By default, the key is set to the value 1, which is **CTRL-A**.

The **-S** and **-T** options are for configuring the YModem file transfer protocol support. See the section on **Catalina YModem Support**.

Use the **-p** and **-s** options to force payload to use a particular port if the auto-detection is not working correctly (or if you have multiple Props connected and need to force payload to use a particular port). The **-p** option is *required* if you are intending to use the interactive mode but are not actually loading a program.

The **-q** option allows programs compiled for a particular line termination style (e.g Windows style, which terminates lines with both a CR and an LF) to be used in the interactive terminal emulator. Multiple **-q** options can be specified - they will all be 'or'ed together to specify the mode.

NOTE: The **-s** option will *always* be required when loading XMM programs on the Hydra and Hybrid, otherwise the second load will be done using the same port as the first load.

The various timing-related options are sometimes required (mostly under Linux) to get the timing right when loading programs. For example, if you cannot get the *first* file to load correctly, try using the **-r** option to force a delay between resetting the propeller and beginning the load. If the first file loads correctly but not the second, try pausing between loads using the **-w** option. If that works, use the **-f** option to find a suitable delay time between loads.

Note that under Linux, it may be necessary on some platforms to adjust the read timeout using the **-t** command line option. For example:

```
payload hello_world -t 2000
```

Interactive Mode

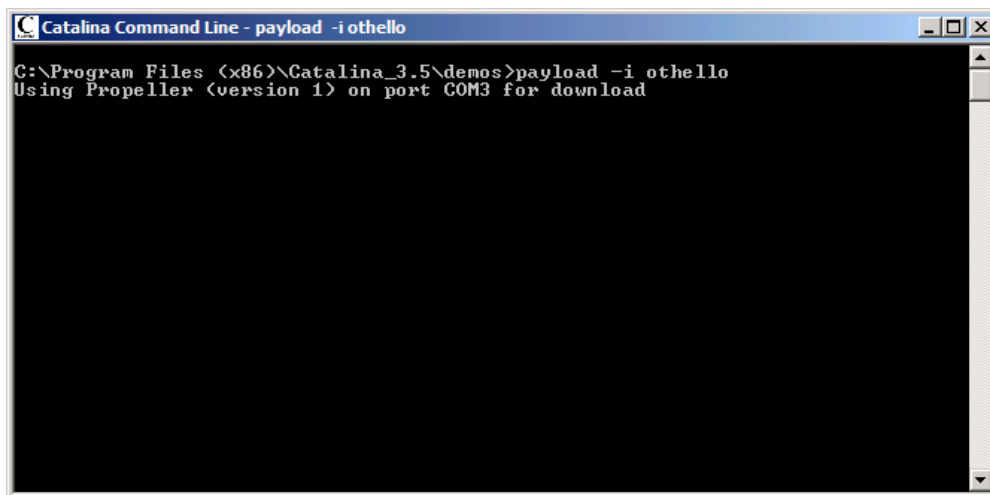
In addition to just loading programs, payload can be used to interact with the Propeller if the loaded program is compiled to use a serial interface (such as **TTY**, **TTY256** or **PC**). Payload offers a simple internal terminal emulator which is adequate for many purposes, and can also invoke an external terminal emulator. One such emulator is provided with Catalina, but other terminal emulators can also be used.

Internal terminal emulation

The internal terminal emulator is invoked by adding **-i** to the payload command. For example:

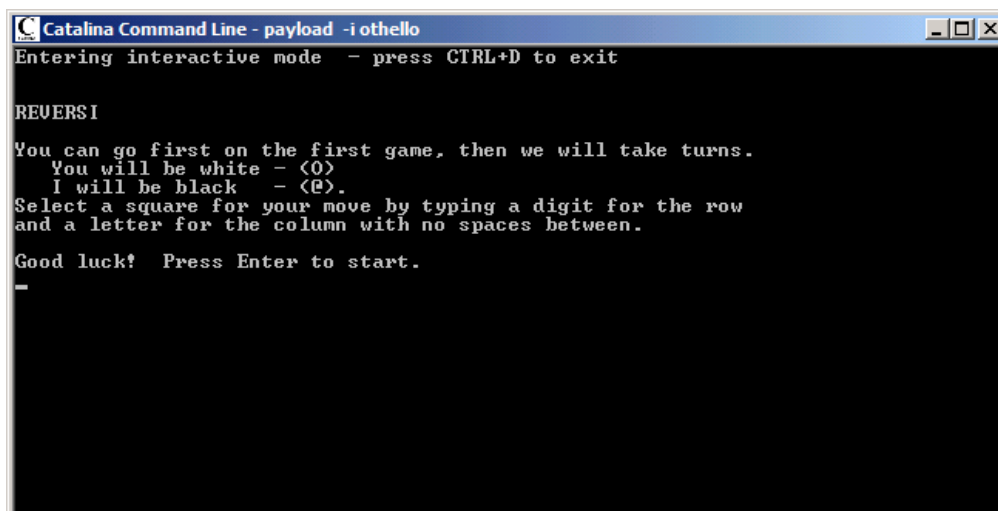
```
catalina othello.c -lc -C PC  
payload othello -i
```

While the program is loading, payload will display its normal messages – i.e. something like the following:



```
Catalina Command Line - payload -i othello
C:\Program Files (x86)\Catalina_3.5\demos>payload -i othello
Using Propeller (version 1) on port COM3 for download
```

Since `-i` is specified, once the program has finished loading, payload will enter interactive terminal mode:



```
Catalina Command Line - payload -i othello
Entering interactive mode - press CTRL+D to exit

REVERSI

You can go first on the first game, then we will take turns.
  You will be white - (O)
  I will be black  - (X).
Select a square for your move by typing a digit for the row
and a letter for the column with no spaces between.
Good luck! Press Enter to start.
-
```

When the program is complete, press CTRL+D to exit the internal interactive mode. Note that you need to press CTRL+D **twice** consecutively to exit payload – pressing it once sends an EOT to the application. If CTRL+D does not work, try CTRL+C, or use the Exit option on the menu that pops up when you press the attention key (CTRL-A by default).

On some Linux systems and with some terminal packages, particularly Gnome Terminal – you must set the terminal defaults correctly so that backspace actually sends a backspace and CTRL+D actually sends an EOT. Check your Linux documentation for more details.

In Gnome Terminal, use the menu command **Edit->Preferences** and select the **Compatibility** tab for your profile (which will be called "Unnamed" if you have not created any specific profile, set the following options:

Backspace key generates: Control-H

Delete key generates: Automatic

The internal terminal emulation is quite simple, and supports a simple subset of VT100 style terminal primitives.

The following key sequences are accepted:

Reset	ESC c
Home	ESC [H
Erase Line	ESC [K
Clear Screen	ESC [2 J
Invisible Curs	ESC [2 5 h
Visible Curs	ESC [2 5 l
Invisible Curs	ESC [? 2 5 h
Visible Curs	ESC [? 2 5 l
Goto Row Col	ESC [<r> ; <c> H

The internal emulator also accepts a Device Status Report request, and responds with a Cursor Position Report:

Device Status Report	ESC [6 n
Cursor Position Report	ESC [<r> ; <c> R

The following key sequences are sent when the corresponding keys are pressed:

↑	ESC O A
↓	ESC O B
→	ESC O C
←	ESC O D
HOME	ESC O w
END	ESC O q
HELP	ESC O p
PREV	ESC O y
NEXT	ESC O s

These primitives are sufficient to use the payload terminal emulator to load and run the **vi** text editor if that program is compiled to use a serial HMI option and the

VT100 option is also specified – **vi** is a full screen text editor which is provided as one of the catalyst demo programs. See the **Catalyst User Manual** for more details.

External terminal emulation

An external terminal emulator can be invoked by adding **-I** to the payload command. The **-I** option requires a terminal name as a parameter. For example:

```
catalina othello.c -lc -C PC
payload othello -I vt100
```

The parameter to the **-I** option is actually the name of a script that payload will execute. On Windows this will be a batch file, on Linux it will be a shell script.

To be used with the **-I** option, the script should accept two parameters – the port name (not the port number) and baud rate to use. Payload will pass the appropriate parameters to the script based on the options passed to **payload** itself.

For example, if the payload command executed was:

```
payload program.bin -p11 -b230400 -I vt100
```

Then after loading the **program.bin** file, the command executed (on Windows) would be:

```
vt100 COM11 230400
```

On Linux, the command would be something like:

```
vt100 /dev/ttyUSB0 230400
```

The script (i.e. on Windows a batch file called **vt100.bat**, or on Linux a shell script called **vt100**) can be used to specify the external terminal emulator to execute, and add any other required parameters.

The following Windows scripts are provided (each one is a batch file):

pc	start comms , specifying PC emulation
vt52	start comms , specifying VT52 emulation
vt100	start comms , specifying VT100 emulation
vt101	start comms , specifying VT101 emulation
vt102	start comms , specifying VT102 emulation
vt220	start comms , specifying VT220 emulation
vt320	start comms , specifying VT320 emulation
vt420	start comms , specifying VT420 emulation
vt100_putty	start PuTTY , specifying VT100 emulation

The following Linux scripts are provided (each one is a shell script):

ansi	start minicom , specifying ANSI emulation
vt100	start minicom , specifying VT100 emulation

vt102 start **minicom**, specifying VT102 emulation

Note that these scripts can also be used independently of payload. They all accept two parameters – the name of the com port to use, and the baud rate. For example:

```
vt100 COM11 115200
```

```
vt102 /dev/ttyUSB1 115200
```

Note that the **comms** program executable is provided as part of the Windows distribution of Catalina. It is not supported on Linux, but the **minicom** terminal emulator is a good alternative.

Also note that when the external comms terminal emulator is in use, the time taken to load and start the emulator can mean the first output of the loaded program might be missed. Adding a short initial delay to the program may be required.

The **comms** program is a full-featured Vtxxx terminal emulator provided with Catalina. It is described in the **Terminal Emulator** document in the Catalina folder *source\comms\doc*. It provides all the functionality of the internal terminal emulator, plus much more:

- Complete VT100/101/102 emulation.

- Font and Color selection.

- Resizable screen buffer (default is 80x24).

- Resizable virtual buffer (default is 1000 lines).

- Full mouse support, including selecting rectangular regions (hold down CTRL while selecting with the mouse).

- Font sizes adjustable on-the-fly (select the Font Sizing option via the main menu, then resize the window).

- Full control of the DTR line (can be used to reset the Propeller).

- Save or load the virtual buffer from a file.

- Print the virtual buffer.

- YModem protocol support.

Both **PuTTY** (on Windows) and **minicom** (on Linux) will have to be installed separately, and the appropriate executable must be somewhere in the current PATH for the scripts to work correctly.

A Note about payload's internal interactive mode under Windows

Windows recently introduced a new console mode that is very buggy. Programs such as **payload** that use console mode can fail unexpectedly if this new console mode is used. This primarily affects payload's internal interactive mode. It requires more investigation, but until Microsoft fixes the issues, there are three possible solutions:

1. Use the external vt100 emulator instead of the internal payload terminal emulator. For instance, instead of saying

```
payload hello_world -i
```

You might say

```
payload hello_world -I vt100
```

However, note that the time taken to start the external terminal emulator means you may miss the initial output of the program. Adding a small delay to program startup may be required.

2. Set the Windows "wrap text output on resize" option. To do this, select **Properties** from the system menu of any console window and in the **Layout** tab, ensure that the option to **Wrap text output on resize** is selected. To make this change permanent, do this in the shortcut that opens the Catalina Command Line window (you will need to restart any open Catalina Command Line windows).
3. Set Windows to use the "legacy console mode". To do this, select **Properties** from the system menu of any console window and in the **Options** tab, ensure that the option to **Use legacy console** is selected. To make this change permanent, make this change on the shortcut that opens the Catalina Command Line window (you will need to restart any open Catalina Command Line windows).

Catalina YModem Support

Both **payload**'s internal terminal emulator and the **comms** external terminal emulator support the YModem serial file transfer protocol, as does the **Catalyst** SD-card based program loader. YModem can therefore be used to transfer arbitrary text or binary files between the Propeller and the Host PC via a serial connection between the two.

Catalyst provides stand-alone YModem **send** and **receive** programs for both the Propeller and the Host PC. You can use **receive** on the Propeller and **send** on the PC to transfer a file from the host to the PC. Or vice-versa.

The syntax of the stand-alone **send** and **receive** programs provided by Catalyst for a Windows or Linux PC host are as follows:

```
send      [-h] [-v] [-d] [-x] [-p PORT] [-b BAUD] [-t TIMEOUT] [-s DELAY] file
receive [-h] [-v] [-d] [-x] [-p PORT] [-b BAUD] [-t TIMEOUT] [file]
```

The stand-alone **send** and **receive** programs for the Propeller are similar except they do not allow the baud rate to be specified on the command line – it must be pre-configured in the platform configuration files (on the Propeller 1, this is the file *Extras.spin*, on the Propeller 2 it is in the *platform.inc* file – e.g. *P2_EDGE.inc*).

Also note that on the Propeller, if a serial HMI option is in use, then the **-h**, **-v** and **-d** options are a bit useless. This is because YModem generally uses the same port as a serial HMI, so the HMI cannot be used at the same time.

Currently, each YModem session only transfers a single file and then terminates. The filename must be specified on the sending end, but is optional on the receiving end – if not provided, the file name specified by the sender will be used.

The file name can be up to 64 characters, and it can include a path – e.g. *myfolder/mysubfolder/myfile.txt*. Such a name would be fine when transferring a file between a Propeller and a Linux host, but not when transferring files between a Propeller and a Windows Host, because Catalyst on the Propeller uses */* as a path separator, but Windows uses ** as a path separator. In such cases, you *must* specify a suitable file name to the receiver as well as the sender. Or only transfer files to and from the current directory on either side so that no path is required.

You can terminate an executing **receive** or **send** program by manually entering two successive **[CTRL-X]** characters.

The default baud rate for all programs is 230400 baud. This is fine on the Propeller 2 but is generally too fast on the Propeller 1, where a baud rate of 115200 should be specified. Also, the Propeller 1's smaller serial buffer sizes and slower serial plugins means the **-s** option must usually be specified for the sender (the **-s** option applies ONLY to the sender). This option does two things:

1. Tells the sender to only send 128 byte blocks, not 1024 byte blocks.
2. Adds a delay of the specified number of milliseconds between each character sent – often **-s0** will work fine, but if not try **-s5**, **-s10** etc.

When the YMODEM program is executing, you may see **C** characters being printed repeatedly in the terminal window – this is normal, and is how the YMODEM send and receive programs synchronize with each other.

The **payload** loader has built-in support for the YModem protocol, which means you use the stand-alone send and receive programs on the Propeller, but on the host you can just use **payload**.

The **payload** loader must be used in *interactive* mode to use YModem, but note that this does not mean that Catalyst has to be using a serial HMI option – it *can* do so, but it does not have to do so. However, note that the **send** and **receive** programs must *not* do so. Typically, they are compiled with the **-C NO_HMI** option, but they may be compiled to use a non-serial HMI instead (e.g. **-C VGA**).

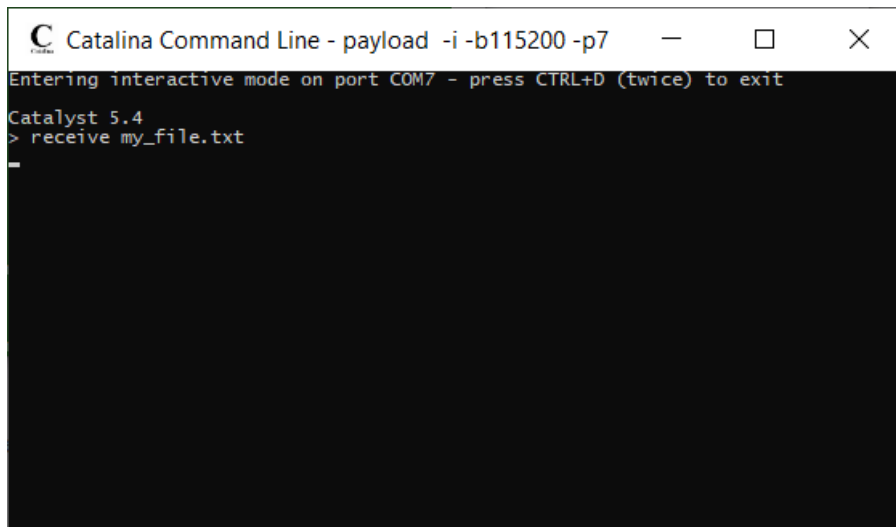
When using a terminal emulator that has YModem support, you typically execute the **send** or **receive** on the Propeller first, and then initiate the YModem transfer in the terminal emulator. For example, with **payload**, and assuming you have a Propeller 1 connected to COM port N, you would typically start **payload** as follows:

```
payload -i -b115200 -pN
```

To receive a command, you would enter a command like the following in **payload** or on your Propeller keyboard:

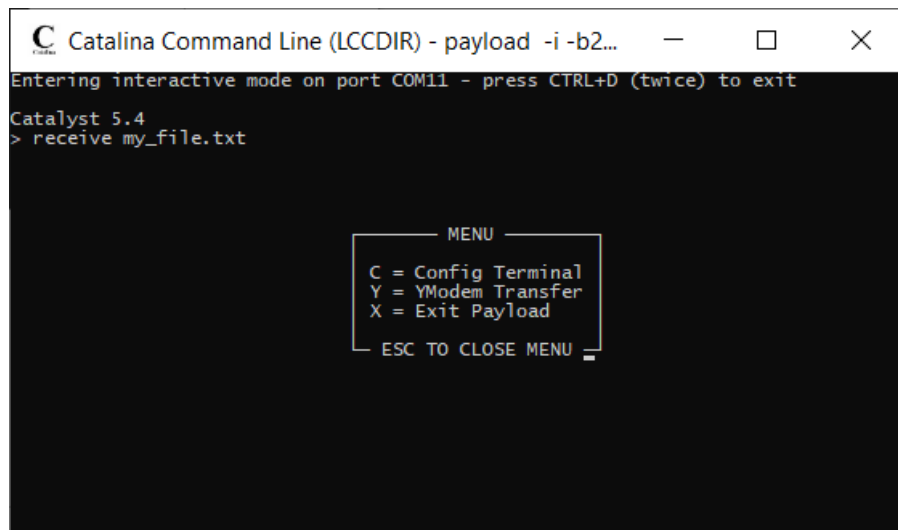
```
receive my_file.txt
```

We will assume you are using a serial HMI, so you would then see a screen similar to the following:

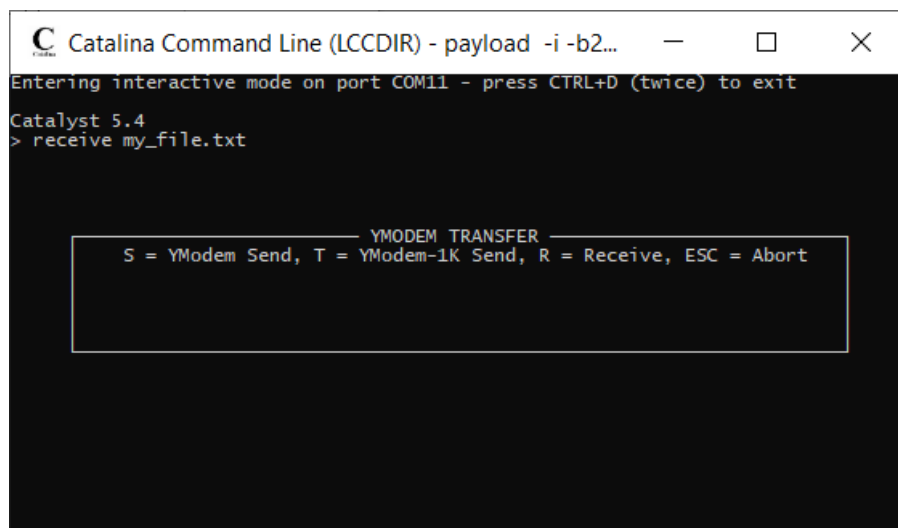


```
Catalina Command Line - payload -i -b115200 -p7
Entering interactive mode on port COM7 - press CTRL+D (twice) to exit
Catalyst 5.4
> receive my_file.txt
```

Now, in payload press the attention key, which by default is set to the key value 1, which corresponds to the key **[CTRL-A]**. You will see a menu, as follows:



Then press **Y** and the YModem Transfer dialog will appear, as follows:



Since we are using a Propeller 1, we cannot use YModem-1K (i.e. 1024 byte block) mode and must press **S** to initiate a YModem 128 byte block transfer. Then fill in the name of the file to send, and press **ENTER**.

If the file transfer completes successfully (it may take some time, depending on the file size), you might see a screen like:

```

Catalina Command Line - payload -i -b115200 -p7
Entering interactive mode on port COM7 - press CTRL+D (twice) to exit
Catalyst 5.4
> receive my_file.txt

      YMODEM TRANSFER
      Enter S for YModem Send, T for YModem-1K Send, R for Receive
      Enter YModem Send Filename
      my_file.txt
      Sending ... 44768 bytes Sent
      Press any key to continue_
  
```

While YModem is quite reliable, serial communications can never be guaranteed, so it is worth checking that the received file size matches the sent file size. YModem is a self-correcting protocol that uses a 16 bit CRC check on each block and re-transmits the block if an error is detected, so if the file size matches it is highly unlikely there will be any errors in the file.

When using a terminal emulator that does *not* have YModem support, you typically execute the **send** or **receive** on the Propeller first, and then terminate the emulator and then execute the host corresponding **receive** or **send** program on the Host command line. We will use **payload**, but without using its built-in YModem support. If N is the number of the COM port to which your Propeller is connected, then to send a file called *my_file.bin* to a Propeller 1:

payload -i -b115200 -pN	<-- executed in command window
receive	<-- executed in payload
[CTRL-D] [CTRL-D]	<-- to exit payload
send my_file.bin	<-- executed in command window

Note that you do not have to use **payload** – other terminal emulators that support YModem may also be used. For example, on Linux the **minicom** terminal emulator can be used. However, note that while the Propeller 2 supports both 1024 byte and 128 byte block YModem transfers, the Propeller 1 only supports 128 byte block transfers, and many YModem implementations assume they can use 1024 byte block transfers (e.g. **Extra PuTTY** and **Tera Term** both assume this, which means they can be used with a Propeller 2, but not a Propeller 1). If you have a Propeller 1, use **payload** or the stand-alone Catalyst YModem **send** and **receive** host programs.

See the **Catalyst User Manual** for more details on Catalyst. And see the **Using the Payload Loader** section of this manual for more details on **payload**.

Building the Payload Loader utilities

Payload is a very flexible loader, and is made even more flexible by it's multi-file load capability. To build the load utilities to be used in multi-file loads, a **build_utilities** batch file is provided, which will interactively ask for details of your platform, and then build some or all of the following utilities:

EEPROM.binary

SRAM.binary

FLASH.binary

MOUSE.binary

On Multi-CPU platforms, each will be appended by an **_n** to indicate the CPU it is compiled for (e.g. **SRAM_1.binary**, **SRAM_2.binary** etc).

For historical reasons, one of the **SRAM.binary** or **FLASH.binary** files will also be copied to **XMM.binary** – depending on which CPU and which type of load you specified as the default (i.e. **SRAM** or **FLASH**).

These utilities allows payload to be used to load programs as follows (the following assumes you have run the *build_utilities* batch file for your platform):

EEPROM.binary – TINY or CMM programs can be loaded to any 32kb EEPROM – they do not need any special compile commands. XMM programs (SMALL or LARGE) can be loaded to EEPROMs of 64 kb or larger, provided they are compiled with the -C EEPROM option. For example:

```
catalina othello.c -lci -C C3
payload EEPROM othello
```

Note that for TINY or CMM programs, the above command has the same effect as:

```
catalina othello.c -lci -C C3
payload -e othello
```

However, neither one is the same as:

```
catalina othello.c -lci -C C3 -C EEPROM
payload EEPROM othello
```

The above commands compile the program to use the EMM target, and the resulting program *must* be loaded using the EEPROM loader. There are reasons why you might want to do this – see the section entitled **EMM Support** for more details.

XMM programs (SMALL or LARGE) can be loaded into EEPROM if compiled with the EEPROM command line option. For example (assuming we built the C3 utilities with a cache size of 1K):

```
catalina othello.c -lci -C C3 -C SMALL -C CACHED_1K -C EEPROM
payload EEPROM othello
```

SRAM.binary – XMM programs (SMALL or LARGE) can be loaded directly into SRAM. For example (assuming we built the C3 utilities with a cache size of 1K):


```
catalina othello.c -lci -C C3 -C SMALL -C CACHED_1K  
payload SRAM othello
```

MOUSE.binary – on the Hydra or Hybrid, loading XMM programs (SMALL or LARGE) must be loaded via the mouse port, using the special mouse loader. For example (assuming we built the C3 utilities with a cache size of 1K) :

```
catalina othello.c -lci -C HYDRA -C SMALL -C CACHED_1K  
payload MOUSE othello
```

FLASH.binary – on platforms with Flash RAM, SMALL XMM programs can be loaded directly into Flash (LARGE programs can only be loaded into Flash if the platform also has some SRAM). For example (assuming we built the **C3** utilities with a cache size of 1K):

```
catalina othello.c -lci -C FLASH -C C3 -C SMALL -C CACHED_1K  
payload FLASH othello
```

or

```
catalina othello.c -lci -C FLASH -C C3 -C LARGE -C CACHED_1K  
payload FLASH othello
```

On platforms with Flash RAM, the **build_utilities** batch file also builds the **Flash_Boot.binary** utility. This utility can be loaded to execute a program already loaded into Flash RAM. If you want this program to execute automatically on Propeller reboot, you can program it into EEPROM.

Finally, to retain compatibility with previous versions of Catalina, the **build_utilities** batch file also creates **XMM.binary** – but this will now simply be a copy of the default XMM loader (i.e. either **SRAM.binary**, **FLASH.binary**, **MOUSE.binary**). Similarly, on the **Hydra** or **Hybrid**, it will also create **Hydra_Mouse.binary** or **Hybrid_Mouse.binary** – but the use of these files is deprecated.

There are examples of using all the payload loader utilities in the *demos\utilities* directory. Refer to the **README.TXT** file in that directory for more details.

Catalina Support for the Propeller

SPIN/PASM Assembler Support

On the Propeller 1, Catalina currently uses **spinnaker** as its default SPIN/PASM assembler. Sources for this are included.

Memory Management Support

Catalina, like most C compilers, has two areas of memory that can be dynamically allocated by the program at run-time - the **stack** and the **heap**, as well as memory that is statically allocated (e.g. used for data and static variables) that is allocated at compile-time.

The stack is usually managed automatically by the compiler - e.g. local variables in a function are dynamically allocated on the stack, and Catalina provides the usual set of standard ANSI C memory management functions for allocating memory on the heap (i.e. **malloc**, **free**, **calloc**, **realloc**). There is also **sbrk** and **_sbrk** which is used by these functions as required to grab chunks of memory from the heap, but it is generally recommended not to use **sbrk** or **_sbrk** directly (they are not part of ANSI C), but instead use the **malloc** functions.

In many cases, this is all a portable C program needs to know about. However, the Propeller architecture adds some complexities to this, especially when XMM RAM is used - this is because C does not provide support for *different types* of memory.

This is not an issue in LMM, CMM or XMM SMALL programs, because both the C stack and the C heap are in Hub RAM - in LMM and CMM programs everything is in Hub RAM, and in XMM SMALL programs everything except code is in Hub RAM. This means that the malloc family of functions are sufficient to allocate RAM dynamically.

However, in a Catalina XMM LARGE program, the C stack is in Hub RAM, but the C heap is in XMM RAM. Since malloc always allocates from the heap, there is no ANSI C means of specifically allocating Hub RAM dynamically, except for instances where the compiler does it for you (such as allocating local variables).

Catalina offers two distinct mechanisms intended to address this issue:

- **hub_malloc** - a set of malloc-like functions that specifically use Hub RAM.
- **alloca** - a function for allocating RAM dynamically on the stack (which is always in Hub RAM).

Each of these is described separately below:

The hub_malloc functions

An additional implementation of malloc type functions has been added to the library which always allocates from Hub RAM. They are only supported in XMM LARGE programs, where the normal **malloc** allocates from the heap, which is in XMM RAM. In other memory models, including XMM SMALL programs, the heap is in Hub RAM, so **hub_malloc** would not provide any benefit over the normal **malloc**.

The following functions are defined in a new include file (*hmalloc.h*), which mirror the usual malloc related function definitions (defined in *stdlib.h*):

```
void    *hub_calloc(size_t _nmemb, size_t _size);
void    hub_free(void *_ptr);
void    *hub_malloc(size_t _size);
void    *hub_realloc(void *_ptr, size_t _size);
```

To support the hub malloc functions, there are also new functions **hbrk()** and **_hbrk()** which are analogous to the usual **sbrk()** and **_sbrk()** functions, but which allocate from Hub RAM instead of XMM RAM.

An example of using the new malloc is provided in *demos\hub_malloc*. The program provided can be compiled to use either standard malloc or the new hub malloc. See the *README.TXT* file in that folder for more details.

An XMM LARGE program can use both **malloc** and **hub_malloc** freely, but other programs should use only one or the other. Also, note that some library functions (notably the stdio and posix thread functions) use the malloc functions internally, and so programs other than XMM LARGE programs should not use the hub malloc functions unless the library is first recompiled to ALSO use hub malloc - this can be done by defining the Catalina symbol **HUB_MALLOC** when compiling the library. This may speed up XMM LARGE programs that make very heavy use of stdio.

Note that using the hub malloc functions reduces the amount of stack space (which is always in Hub RAM in every memory mode) available to the program.

For an example of using **HUB_MALLOC** to compile the library, see the *build_all* scripts provided in the folder *demos\catalyst\catalina*. Using this option has been tested, but since it reduces the available Hub RAM for ALL programs it is not enabled by default.

Note that the two types of allocated memory can be used freely without knowing what type of memory it is, but that memory allocated using the normal **malloc** or **calloc** functions MUST be re-allocated or freed using **realloc** and **free**, and memory allocated using the **hub_malloc** or **hub_calloc** functions MUST be reallocated or freed using **hub_realloc** and **hub_free**.

The alloca function

Catalina now supports an **alloca** function. This function is supported in all memory models, and on both the Propeller 1 and Propeller 2. Note that **alloca** is not part of the ANSI C standard, but it is commonly available in C compilers.

The **alloca** function returns a pointer to a dynamically allocated area of stack space that is valid until the function in which it is called returns to its caller. The **alloca** function is built-in to the Catalina compiler, but is defined in *alloca.h* as if it was a normal C function with the following function prototype:

```
void *alloca (size_t __size);
```

For more details on **alloca**, see <https://linux.die.net/man/3/alloca>

The main advantage of **alloca** is that it is a very small and efficient alternative to **malloc**. It is particularly suited to C on the Propeller, where the heap and stack can use different types of RAM. The C language does not cater very well for architectures that can have different types of RAM.

Note that there are limitations of using **alloca**, which are described in the link above - the main one is that (unlike memory allocated using **malloc**) memory allocated using **alloca** must not be referenced once the function in which it was allocated returns to

its caller. However, it is worth remembering that such memory allocated in the C **main** function will remain valid for the duration of the program. But this also illustrates another limitation of **alloca** - which is that there is no way to DE-allocate such memory once allocated other than by exiting the function in which it is allocated.

The **alloca** function is particularly useful on the Propeller. Consider an XMM LARGE program. In these programs the stack is in Hub RAM, but the heap is in XMM RAM, and the **malloc** functions always operate only on the heap. While the **hub_malloc** functions can do this, these functions (like **malloc**) are large in code size and also very slow when compared to **alloca**.

For an example of using **alloca**, see the *demos\alloca* folder.

Specifying the heap size

When using the standard C dynamic memory management functions that use the heap - i.e. **malloc()**, **realloc()**, **calloc()** and **free()** - the **catalina**, **catbind** and **bcc utilities** all have a command line option for specifying the maximum address to use for the heap. The **-H** option accepts an address parameter and can be used to specify the maximum address that will be used by the heap. In all memory modes except **LARGE** mode, the heap and stack share Hub RAM, with the heap growing upward from the highest used low Hub address, and the stack growing downward from the lowest used high hub address. This means they can eventually overlap, with potentially disastrous consequences. Even in **LARGE** programs, it may be desirable to limit the size of the heap to less than the entire available RAM - e.g. to reserve part of the upper RAM for other purposes.

The **-H** option allows this to be avoided, by limiting the growth of the heap. The program may run out of heap space, but the failure is detectable (e.g. **malloc** will return an error if there is no more heap space). The required amount of stack space can be determined by printing the current stack pointer at various suitable points in the program - below is a macro that uses inline PASM to do this, and a trivial program that uses it. This program will work in any memory model on any Propeller:

```
// this handy macro returns the current stack pointer
// in any memory model on the P1 or P2 ...
#define SP_PASM( \
    "#ifdef COMPACT\n" \
    "    word I16B_PASM\n" \
    "#endif\n" \
    "    alignl\n" \
    "    mov r0, SP\n")

void main() {
    printf("SP=0x%06X\n", SP);
    while(1);
}
```

Suppose on a Propeller it was known that the stack could grow down to 0x6000 - then it might be appropriate to specify **-H 0x6000** to prevent the heap ever growing large enough to overwrite the stack. The parameter can be specified as decimal

(including an optional 'k' or 'm' suffix) or as hexadecimal (using the format \$XXXXXX or 0xXXXXXX).

For example, to ensure the heap never grows above 24k, leaving the top 8k for buffers and stack space, use a command like:

```
catalina prog.c -lc -H 24576
```

or

```
catalina prog.c -lc -H 24k
```

or

```
catalina prog.c -lc -H 0x6000
```

The **-H** option can be used on the Propeller 1 or 2. In all modes except **LARGE** mode the address refers to a Hub address. It can also be used in **LARGE** mode, where the heap is in XMM RAM, but the address refers to an XMM RAM address. This could be used (for example) to reserve an upper area of XMM RAM for other uses, such as for a buffer. However, note that the start address of the XMM RAM can vary from platform to platform, so check the **XMM_RW_BASE_ADDRESS** in the various platform configuration files.

Note that **-H** only affects the heap used by the standard C memory management functions - it does not affect the **hub_malloc** or **alloca** functions.

Floating Point Support

Catalina provides several options for 32 bit IEEE 754 floating point support.

The fundamental 32 bit floating point operations (i.e. addition, subtraction, multiplication, division, comparison and conversion between floats and other data types) are built into the default Catalina LMM Kernel, and incur no additional overhead during execution. This means that the fundamental floating point operations are as fast as the equivalent PASM operations (in fact they *are* the equivalent PASM operations).

For support of the standard C89 math library functions (sin, cos, tan, exp, pow etc), Catalina provides options similar to those provided by the Parallax Float32 libraries:

- **Float32_B** and **Float32_A**. All the math functions supported by Float32Full are implemented using two cogs, with software emulation of other required functions not implemented in Float32Full (e.g. sinh, cosh tanh). This is the best solution for programs that can spare two cogs. Requires that the **libmb** library be used (i.e. use command line option **-lmb**).
- **Float32_A**. All the math functions supported by Float32A are implemented using one cog, with software emulation of other required functions not implemented by Float32A (e.g. log, exp, pow, sinh, cosh, tanh). This is a good solution for programs that can only spare one cog. Requires that the **libma** library be used (i.e. use command line option **-lma**).

- **Software.** All the math functions are emulated in software. This is a good solution for programs that cannot spare any cogs, but it may require more RAM, and is also 3 to 4 times slower. Requires that the **libm** library be used (i.e. use command line option **-lm**).

All options are transparent to the C program that uses them. The choice depends mainly on how much RAM and how many spare cogs are available.

Custom combinations can also be created as a plugin and used in a specific Catalina target.

HMI Support

For platforms with keyboard and mouse inputs, and a VGA or TV output, various HMI (Human Machine Interface) configurations are provided:

Built-in device support:

- Display
 - o High-resolution VGA (40x24 to 125x64 chars);
 - o Low resolution VGA (32x16 chars);
 - o High resolution TV (40x30 chars);
 - o Low resolution TV (40x13 chars);
- Keyboard;
- Mouse.

Terminal emulator support ¹²:

- **PC** terminal emulator (supports using the PC display and keyboard);
- **TTY** terminal emulator (similar to PC, but uses one less cog and doesn't support proxy drivers on multi-CPU systems);
- **TTY256** terminal emulator (similar to TTY, but with 256 byte transmit and receive buffers);
- **PROPTERMINAL** ¹³ (supports using the PC display, keyboard and mouse via the PropTerminal program).

The HMI configuration used by a Catalina program is determined by the target selected when building the program, as well as by command line options. Note that not all configurations are supported on all platforms. For example the **Hybrid** supports only TV configurations. As far as possible, Catalina attempts to detect if an unsupported HMI configuration is specified.

¹² On some platforms, such as the Hybrid and Hydra, the serial or USB port cannot be used at the same time as the SRAM or SD card. This means that on these platforms, terminal emulator support is only available to LMM programs that do not use the SD card.

¹³ For more information on **PropTerminal**, and to download a binary version, see the Insonix web site <http://www.insonix.ch/propeller/> (press the English flag for an English translation of the site!)

In this version of Catalina, all the different HMI options are ANSI compliant (by default) regarding how they handle characters on input and output. If it is necessary to change this behavior, the following command line options can be used:

CR_ON_LF	Translate LF to CR LF on output
NO_CR_TO_LF	Disable translation of CR to LF on input
NON_ANSI_HMI	Disable ANSI compliance in HMI (revert to previous Catalina behavior)

More information on the HMI options supported by the standard targets is provided in the **Catalina Targets** section later in this document.

Catalina also provides proxy HMI drivers, which on multi-CPU systems allows one CPU to use the HMI devices physically connected to another CPU. This is further described in the **Multi-CPU System Support** section later in this document.

Each Catalina target defines a default HMI for each supported platform, with a standard set of options, and there are various command line options that can be used to select or configure the HMI for a particular program if the default is not appropriate. See the section **Default Target Configuration Options** for more detail.

The choice of HMI options depends largely on the hardware available on the Propeller platform, and how much RAM is required for the C program. Additional customized HMI configurations can be created if required.

If a particular platform does not have some of the HMI devices (or a particular C program does not need them) then HMI configuration options can be used to specify that unnecessary drivers are not started. However, the RAM used by drivers is reclaimed anyway once they are loaded, and made available as heap and stack space for the Catalina program (see the **Startup and Memory Management** section later in this document) – so doing this may save cogs, but may not actually save any RAM.

Most of the HMI functions are straightforward, and are typically very similar to the equivalent functions provided by the individual underlying screen, keyboard or mouse drivers – which are often the Parallax standard drivers. See the individual drivers for details. However, a few of the screen functions are added by the HMI plugin itself, and may require a little more explanation:

All the different HMI options provide exactly the same interface to Catalina C programs, using the standard C streams – i.e. **stdin**, **stdout** & **stderr**. C functions are also provided for using the underlying HMI drivers more directly – these will be familiar to many Propeller users.

Keyboard functions

```
int k_present();
```

This function returns one if a keyboard is detected, or zero otherwise. Not supported (always returns one) on the PC, TTY, TTY256. PROPTERMINAL or proxy HMI drivers.


```
int k_get();
```

This function returns the next key from the keyboard, or zero if no key is available. To check if a key is available before calling, use **k_ready**. To wait for a key, use **k_wait** or **k_new** instead.

```
int k_wait();
```

This function returns the next key from the keyboard. If no key is currently available, it waits for the next key.

```
int k_new();
```

This function deletes any keys stored in the keyboard buffer, then waits for the next key.

```
int k_ready();
```

This function returns one if a key is available, or zero otherwise.

```
int k_clear();
```

This function clears any keys stored in the keyboard buffer but not yet read.

```
int k_state(int key);
```

This function returns the state of the specified key (one if pressed, zero if not). Not supported (always returns zero) on the PC, TTY, TTY256. PROPTERMINAL and proxy HMI drivers.

Mouse functions

```
int m_present();
```

This function returns one if a mouse is detected, or zero otherwise. Always returns one on the PROPTERMINAL HMI drivers, and not supported (returns zero) on the TTY, TTY256 and PC HMI drivers.

```
int m_button (unsigned long b);
```

This function returns the current state of mouse button b (0, 1 or 2).

```
int m_buttons();
```

This function returns the current state of all mouse buttons as a set of bits.

```
int m_abs_x();
```

This function returns the current absolute x value of the mouse.

```
int m_abs_y();
```

This function returns the current absolute y value of the mouse.

```
int m_abs_z();
```

This function returns the current absolute z value of the mouse.

```
int m_delta_x();
```

This function returns the current delta x value of the mouse.

```
int m_delta_y();
```

This function returns the current delta y value of the mouse.

```
int m_delta_z();
```

This function returns the current delta z value of the mouse.

```
int m_reset();
```

This function resets the mouse, and sets the x,y,z values to zero.

```
void m_bound_limits(int xmin, int ymin, int zmin,
                   int xmax, int ymax, int zmax);
```

This function sets the minimum and maximum bounding limits for each of the x, y and z axes.

```
void m_bound_scales (int xscale, int yscale, int zscale);
```

This function sets the bounding scales for each of the x, y and z axes.

```
void m_bound_preset (int xpreset, int ypreset, int zpreset);
```

This function sets the preset bound coordinates of the x, y and z axes.

```
int m_abs (int value);
```

This function is used internally. It is made visible only because it is visible in the standard Parallax mouse drivers, and therefore some users of those drivers may have written programs that depend on it.

```
int m_limit (int i, int value);
```

This function is used internally. It is made visible only because it is visible in the standard Parallax mouse drivers, and therefore some users of those drivers may have written programs that depend on it.

```
int m_bound (int i, int delta);
```

This function is used internally. It is made visible only because it is visible in the standard Parallax mouse drivers, and therefore some users of those drivers may have written programs that depend on it.

```
int m_bound_x();
```

This function returns the current x bound of the mouse.

```
int m_bound_y();
```

This function returns the current y bound of the mouse.

```
int m_bound_z();
```

This function returns the current z bound of the mouse.

Screen functions

```
int t_geometry();
```

This function returns the screen geometry (as columns * 256 + rows).
Not supported (returns zero) for the TTY and TTY256 HMI driver.

```
int t_char(unsigned curs, unsigned ch);
```

This function writes a character to the current cursor location. Cursor 0 or 1 can be used.

```
int t_string(unsigned curs, char *str);
```

This function writes a zero terminated string to the current cursor location. Cursor 0 or 1 can be used.

```
int t_integer(unsigned curs, int val);
```

This function converts its signed integer argument to a string and writes it to the current cursor location. Cursor 0 or 1 can be used.

```
int t_unsigned(unsigned curs, unsigned val);
```

This function converts its unsigned integer argument to a string and writes it to the current cursor location. Cursor 0 or 1 can be used.

```
int t_float(unsigned curs, float val, int digits);
```

This function converts its floating point argument to a string and writes it to the current cursor location. Cursor 0 or 1 can be used. Not supported when using **libci** or **libcix**

```
int t_hex(unsigned curs, unsigned val); 14
```

This function converts its unsigned integer argument to a hexadecimal string (containing characters '0' to '9' and 'A' to 'F') and writes it to the current cursor location. Cursor 0 or 1 can be used.

```
int t_bin(unsigned curs, unsigned val); 15
```

This function converts its unsigned integer argument to a binary string (containing only characters '0' and '1') and writes it to the current cursor location. Cursor 0 or 1 can be used.

```
int t_setpos(unsigned curs, unsigned cols, unsigned rows);
```

This function sets the position of a cursor – either cursor 0 or cursor 1. The selected cursor is moved to the position specified.

```
int t_getpos(unsigned curs);
```

This function gets the position of a cursor – either cursor 0 or cursor 1. Returns the current position of the selected cursor (as column * 256 + row).

```
int t_mode(unsigned curs, unsigned mode);
```

This function sets the wrap/scroll and cursor modes of a cursor – either cursor 0 or cursor 1. All the HMI drivers implement two independent cursors. For all drivers except the high resolution VGA driver, the mode of each cursor is a byte with bit values as follows:

¹⁴ Currently not supported in all drivers due to space limitations. Instead, it is supported as a C library function. This is only significant if you intend to write PASM code that expects to call the driver service directly.

¹⁵ Ditto.

- bit 0:** 0 – cursor invisible (cursor 1 only)
1 – cursor is a visible block (cursor 1 only)
- bit 3:** 0 – cursor wraps at end of screen
1 – screen scrolls end of screen

For these drivers cursor 0 is always invisible. For a visible cursor, use cursor 1. Cursor 1 (when visible) is always a blinking block.

For the high resolution VGA driver, the mode of each cursor is a byte with bit values as follows:

- bits 0,1:** 00 – cursor invisible
01 – cursor visible (unblinking)
10 – cursor visible (slow blink)
11 – cursor visible (fast blink)
- bit 2:** 0 – cursor is a block
1 – cursor is an underscore
- bit 3:** 0 – cursor wraps at end of screen
1 – screen scrolls end of screen

```
int t_scroll(unsigned count, unsigned first, unsigned last);
```

This function scrolls the screen up a specified number of lines. The count is the number of lines to scroll, and first and last are the first row to scroll and the last row to scroll. .

```
int t_color(unsigned curs, unsigned color);
```

This function sets the screen color. For all HMI drivers except the high resolution VGA driver, color is a number 0 to 7 that refers to an entry in a color palette built in to the driver, and each character cell can have its color set independently of the others. The new color is applied to any new characters output to the screen – to apply the specified color to the whole screen, you can clear the screen.

For the high resolution VGA driver, color works differently – it is specified as a 16 bit number (as $bg * 256 + fg$) with each color being specified as 8 bits with the bit values RRGGBB00 – i.e. 2 bits each for red, green and blue. The color is applied to the whole row indicated by the specified cursor (and only to that row – i.e. it does not apply if the cursor is then moved to a new row).

Utility functions

```
int t_printf (char *fmt, ...);
```

This function works very much like the standard C function **printf**. It requires significantly less space, but it supports only a few formatting options:

%c print a character

%d print an integer as a decimal number

%f print a floating point value. The **f** can be preceded by an optional “precision” which is a single number from 0 to 9 indicating the number of digits to follow the decimal point – for example **%3f** prints 3 digits after the decimal point. Not supported when using **libci**.

%s print a string

%x print an integer as a hexadecimal number

Any other character is printed as it appears. For example:

```
t_printf("char = %c\nstr = %s\nfloat = %3f\n", c, str, f);
```

CGI (Computer Graphics Interface) Support

Catalina provides a computer graphics plugin and C library equivalent to the Parallax standard Graphics object.

All that is required to use the computer graphics plugin is to link with the **libgraphics** library. The appropriate drivers will be loaded automatically if a Catalina program uses this library.

For an example, go to the *demos\graphics* sub-directory and execute the **build_all** script, specifying your platform. It will execute commands similar to:

```
catalina -lci -lgraphics graphics_demo.c -C NO_HMI
```

The **NO_HMI** option is generally required when using the graphics plugin because some of the normal HMI drivers conflict with the corresponding graphics drivers. However, if your program uses HMI drivers that do not conflict with the graphics drivers (such as the **PC** HMI drivers) then these can be included along with the graphics drivers - provided there are enough free cogs.

The graphics plugin can be used in both LMM and XMM modes.

The basic parameters of the graphics plugin, such as the resolution, are set in *Catalina_Common.spin*:

```
X_TILES    = 16      ' Tiles are 16 by 16, so default X resolution is 256
Y_TILES    = 12      ' Tiles are 16 by 16, so default Y resolution is 192
```

The following symbols can be defined on the command line to modify the behavior of the graphics plugin:

NO_INTERLACE	Set the TV driver to NO_INTERLACE mode. In the current release, only the TV driver is supported. VGA driver support will be added in a subsequent release.
NTSC	Sets the TV driver to NTSC (rather than PAL) mode.
DOUBLE_BUFFER	Allocates two graphics buffers. All updates are performed on one graphics buffer, which is then copied to the display graphics buffer. This results in smoother (but slower) video. However, double buffering requires significantly more Hub RAM than single buffering.
NO_HMI	CGI drivers cannot generally be used in conjunction with the equivalent HMI drivers (e.g. for mouse or keyboard). To avoid this conflict, specify NO_HMI .
NO_KEYBOARD	do not load the CGI keyboard driver.
NO_MOUSE	do not load the CGI mouse driver.

For example:

```
catalina -lci -lgraphics graphics_demo.c -C NO_HMI -C DOUBLE_BUFFER
```

The *graphics_demo.c* program in the *demos\graphics* folder is a faithful copy of the standard Parallax demo program, and includes examples of the use of most graphics functions.

Many of the graphics functions take a pointer to a **g_var** structure. This structure should be allocated as a local variable to ensure that all variables are allocated in Hub RAM, even when we are using the XMM memory model. Normally, programs do not need to know the internal details of this structure – they simply allocate a local instance (which ensures it is in Hub RAM) and then call the **g_setup** function to initialize it. For example:

```
main() {
    g_var gv;
    g_setup(&gv, 120, 80, 0);
    ...
}
```

Some of the graphics library functions are used to retrieve information about the underlying CGI driver. These functions do not require a pointer to a **g_var** structure:

```
int cgi_x_tiles();
    Get the value of X_TILES (X resolution in tiles of 16 pixels).

int cgi_y_tiles();
    Get the value of Y_TILES (Y resolution in tiles of 16 pixels)

void *cgi_display_base();
```

Get address of underlying CGI display. Note that this is always a Hub RAM address. The bitmap data will be:

$(x_tiles * y_tiles)$ tiles, or
 $((x_tiles * y_tiles) * 16 * 16 * 2) / 8$ bytes.

```
void *cgi_bitmap_base(int double_buffer);
```

Get the address of the underlying CGI bitmap to draw on. We must tell this function if we are double buffering. Note that this is always a Hub RAM address. The bitmap data will be:

$(x_tiles * y_tiles)$ tiles, or
 $((x_tiles * y_tiles) * 16 * 16 * 2) / 8$ bytes.

```
void *cgi_screen_data(int double_buffer);
```

Get address of underlying CGI screen data. We must tell this function if we are double buffering. Note that this is always a Hub RAM address. You must provide x and y size. The screen data will be $(x_tiles * y_tiles)$ words.

```
void *cgi_color_data(int double_buffer);
```

Get address of underlying CGI color data. We must tell this function if we are double buffering. Note that this is always a Hub RAM address. You must provide x and y size. The colors data will always be 64 longs

The remaining graphics library functions emulate the operations of the standard Parallax graphics object. These functions all require a pointer to a **g_var** structure:

```
void g_setup(g_var *gv, int x_org, int y_org, int double_buffer);
```

Set bitmap parameters:

<code>x_org</code>	relative-x center pixel
<code>y_org</code>	relative-y center pixel
<code>double_buffer</code>	true if double buffering

```
void g_clear(g_var *gv);
```

Clear either the display (if not double buffering), or the double buffer bitmap (if double buffering).

```
void g_copy(g_var *gv, void *bitmap_base);
```

Copy either the specified bitmap, or (if NULL) the double buffer bitmap to the display (use for flicker-free display).

```
void g_color(g_var *gv, int color);
```

Set pixel color to two-bit pattern:

<code>color</code>	color code in bits[1..0]
--------------------	--------------------------

```
void g_width(g_var *gv, int width);
```

Set pixel width. Actual width is $w[3..0] + 1$:

width 0..15 for round pixels, 16..31 for square pixels

```
void g_colorwidth(g_var *gv, int color, int width);
```

Set pixel color and width.

color color code in bits[1..0]

width 0..15 for round pixels, 16..31 for square pixels

```
void g_plot(g_var *gv, int x, int y);
```

Plot point:

x, y endpoint

```
void g_line(g_var *gv, int x, int y);
```

Draw a line to point:

x, y endpoint

```
void g_arc(g_var *gv, int x, int y, int xr, int yr, int angle, int
anglestep, int steps, int arcmode);
```

Draw an arc:

x, y center of arc

xr, yr radii of arc

angle initial angle in bits[12..0] (0..\$1FFF = 0°..359.956°)

anglestep angle step in bits[12..0]

steps number of steps (0 just leaves (x,y) at initial arc position)

arcmode 0: plot point(s)
1: line to point(s)
2: line between points
3: line from point(s) to center

```
void g_vec(g_var *gv, int x, int y, int vecscale, int vecangle, void *
vecdef_ptr);
```

Draw a vector sprite:

x, y center of vector sprite

vecscale scale of vector sprite (\$100 = 1x)

vecangle rotation angle of vector sprite in bits[12..0]

vecdef_ptr address of vector sprite definition

The Vector sprite layout in memory is as follows:

```
word $8000|$4000+angle 'vector mode + 13-bit angle
word length '(mode: $4000=plot, $8000=line) 'vector length
```



```

...                               'more vectors
...                               'end of definition
word 0
void g_vecarc(g_var *gv, int x, int y, int xr, int yr, int angle, int
vecscale, int vecangle, void * vecdef_ptr);

```

Draw a vector sprite at an arc position:

x, y	center of arc
xr, yr	radii of arc
angle	angle in bits[12..0] (0..\$1FFF = 0°..359.956°)
vecscale	scale of vector sprite (\$100 = 1x)
vecangle	rotation angle of vector sprite in bits[12..0]
vecdef_ptr	address of vector sprite definition

```
void g_pix(g_var *gv, int x, int y, int pixrot, void *pixdef_ptr);
```

Draw a pixel sprite:

x, y	center of vector sprite
pixrot	0: 0°, 1: 90°, 2: 180°, 3: 270°, +4: mirror
pixdef_ptr	address of pixel sprite definition

The Pixel sprite layout in memory is as follows:

```

word    'word align, express dimensions and center, define pixels
byte    xwords, ywords, xorigin, yorigin
word    %xxxxxxxx, %xxxxxxxx
word    %xxxxxxxx, %xxxxxxxx
word    %xxxxxxxx, %xxxxxxxx
...
void g_pixarc(g_var *gv, int x, int y, int xr, int yr, int angle, int
pixrot, void *pixdef_ptr);

```

Draw a pixel sprite at an arc position:

x, y	center of arc
xr, yr	radii of arc
angle	angle in bits[12..0] (0..\$1FFF = 0°..359.956°)
pixrot	0: 0°, 1: 90°, 2: 180°, 3: 270°, +4: mirror
pixdef_ptr	address of pixel sprite definition

```
void g_text(g_var *gv, int x, int y, void *string_ptr);
```

Draw text:

x, y	text position (see g_textmode() for sizing and justification)
string_ptr	address of zero-terminated string (it may be necessary to call finish immediately afterwards to prevent subsequent code from clobbering the string as it is being drawn)

```
void g_textarc(g_var *gv, int x, int y, int xr, int yr, int angle,
               void *string_ptr);
```

Draw text at an arc position:

x, y	center of arc
xr, yr	radii of arc
angle	angle in bits[12..0] (0..\$1FFF = 0°..359.956°)
string_ptr	address of zero-terminated string (it may be necessary to call finish immediately afterwards to prevent subsequent code from clobbering the string as it is being drawn)

```
void g_textmode(g_var *gv, int x_scale, int y_scale, int spacing, int
                justification);
```

Set text size and justification:

x_scale	x character scale, should be 1+
y_scale	y character scale, should be 1+
spacing	character spacing, 6 is normal
justification	bits[1..0]: 0..3 = left, center, right, left bits[3..2]: 0..3 = bottom, center, top, bottom

```
void g_box(g_var *gv, int x, int y, int box_width, int box_height);
```

Draw a box with round/square corners, according to pixel width

x,y	box left, box bottom
-----	----------------------

```
void g_quad(g_var *gv, int x1, int y1, int x2, int y2, int x3, int y3,
            int x4, int y4);
```

Draw a solid quadrilateral. Vertices must be ordered clockwise or counter-clockwise.

x1, y1 .. x4, y4	vertices of the quadrilateral
------------------	-------------------------------

```
void g_tri(g_var *gv, int x1, int y1, int x2, int y2, int x3, int y3);
```

Draw a solid triangle.

x1, y1 .. x3, y3	vertices of the triangle
------------------	--------------------------

```
void g_finish();
```

Wait for any current graphics command to finish. Use this to insure that it is safe to manually manipulate the bitmap

There are also a full set of mouse functions in this library - these are functionally equivalent to the normal Catalina (or Parallax) mouse functions already described (in the section title **Mouse functions**) except they are prefixed by **gm_** instead of **m_** (e.g. **gm_abs_x()**). Note that programs **MUST** use the graphics versions of the mouse functions if you use the graphics library, not the HMI versions. Also, programs must be sure to call one of the two functions **gm_reset()** or **gm_present()** before

calling any other mouse function, since they automatically initialize the driver (and the other functions do not).

Read the **README.Graphics** file in the *demos\graphics* directory for more details.

VGI (Virtual Graphics Interface) Support

Catalina provides a virtual graphics plugin and C library that is largely compatible with the graphics library described in the previous section. The **Virtual** (or **VGA** or **Vector**) Graphics plugin is intended for use on high resolution VGA displays.

All that is required to use the plugin is to link with the **libvgraphic** library. The appropriate drivers will be loaded automatically if a Catalina program uses this library.

For an example, go to the *demos\vgraphics* sub-directory and execute the **build_all** script, specifying your platform. It will execute commands similar to:

```
catalina -lci -lvgraphic graphics_demo.c -C NO_HMI
```

The **NO_HMI** option is generally required when using the virtual graphics library because some of the normal HMI drivers conflict with the graphics drivers. However, if your program uses HMI drivers that do not conflict with the graphics drivers (such as the **PC** HMI drivers) then these can be included along with the graphics drivers - provided there are enough free cogs.

The virtual graphics library can be used in LMM, CMM and XMM modes (although the performance in XMM mode is very slow!).

The main differences between the virtual graphics library and the graphics library is in the way the library is initialized – the **g_setup**, **g_copy** and **g_move** functions are different, and there is an additional **g_db_setup** function to set up double buffering. Refer to the header file *catalina_vgraphics.h* for more details, or study the differences in the *graphics_demo.c* program given in the *demos\vgraphics* folder.

The following command line options can be used to modify the behavior of the virtual graphics plugin:

NO_KEYBOARD	do not load the vgraphics keyboard driver
NO_MOUSE	do not load the vgraphics mouse driver
VGA_640	resolution is 640 x 480. This is the default.
VGA_800	resolution is 800 x 600.
VGA_1024	resolution is 1024 x 768.
VGA_1152	resolution is 1152 x 864.
VGA_2_COLOR	color depth is 1 bit (2 colors).
VGA_4_COLOR	color depth is 2 bits (4 colors).
DOUBLE_BUFFER	enable double buffering (smoother graphics).

For example:

```
catalina -lci -lvgraphics graphics_demo.c -C NO_HMI -C VGA_800
```

Read the **README.TXT** file in the *demos\lvgraphics* directory for more details.

Multi-Processing Support

Catalina offers extensive support for three different types of multi-processing on the Propeller:

Multi-Cog	A C program can execute C functions on multiple cogs.
Multi-Thread	A C program can create multiple threads of execution that can all run on the same cog, or on different cogs.
Multi-Model	Separate C programs, perhaps using memory models (such as NATIVE and COMPACT) can be executed on separate cogs. This technique can also be used to create dynamically loaded overlays.

Multi-processing and plugins

In a Catalina multi-processor environment, there is still only ever one registry and one common set of plugins. The registry and the plugins are designed to be accessible to all C code no matter how they were loaded and started. However, note that it is always the responsibility of the *main* C program to specify the correct plugins to be used by all other code, even if it does not need those plugins itself.

For instance, if a C program dynamically loads code to execute on another cog, and that code uses floating point, then it will fail **UNLESS** the main program loads a floating point plugin – i.e. **UNLESS** the main program is linked with one of the options **-lma**, **-lmb** or **-lmc**. If it is instead linked with just **-lm**, or not linked with a floating point option at all, then the main program will *not* load the floating point plugin that the dynamically loaded kernel requires.

Multi-Cog Support

Catalina supports running C code on all available cogs. When a C program is started, it is being executed within the cog running the kernel. However, a new kernel can be loaded and run on any available cog, executing a C function.

The C function to be executed should meet the following criteria:

- It must be declared as a void function with no arguments. All communication between C functions running on different cogs must be done via global variables.
- It must be allocated a dedicated stack.
- It should use locks (where necessary) to prevent contention when accessing global variables, or library functions. The nature of the Propeller means that accesses to basic data types (char, int, long, float, pointers, etc) are atomic and do not require locks – but access to more complex data types (e.g. structures, linked lists etc) may need to be protected. Also, calls to library functions which access such complex data types (e.g. **malloc**) will also need to be protected.

- It should never return. If the function needs to terminate it should use **_cogstop** function. To determine its own cog id to stop, it can use the **_cogid** function.

More details on all the Catalina cog functions are given in the section on **Plugin Support**. Here, we will discuss only the creation of a C function on a separate cog.

For example, suppose we have the following C function:

```
void cog_function(void) {
    int me = _cogid();
    ...
    _cogstop(me);
}
```

Starting this function on another cog is done using the **_coginit()** function to start a special dynamically loadable kernel, with initialization data that specifies the C function to be executed.

Since this process can be complex, the details are often wrapped up in a utility function. An example of such a utility function is provided in the *demos/multicog* sub-directory.

This directory contains complete working examples of various multi-cog programs, which make use of the following function, defined in *utilities.h*:

```
int C_coginit(void func(), unsigned long *stack);
```

To use this function, define sufficient stack space, then pass the address of the function to be started, and the address of the TOP of the stack space allocated for it. The function returns the cog allocated to the function or -1 if there is any error starting it. For example, to start the example **cog_function** defined above:

```
int cog;
long cog_stack[100];
cog = C_coginit(&cog_function, &cog_stack[100]);
```

For more details on this process, see the implementation of **C_coginit** in *utilities.c*.

For more details on the **spinc** utility used by the examples, refer to the section on using PASM with Catalina.

Multi-Threading Support

Catalina provides a multithreading library, similar to but much more efficient than, the POSIX **pthread**s library (which it also supports – see later in this document). To use this library, simply compile your program with the **libthreads** library. The multithreading version of the kernel will be included automatically.

For an example, go to the *demos/multithread* sub-directory and execute the **build_all** script, specifying your platform. It will execute commands similar to:

```
catalina -p2 -lci -lthreads dining_philosophers.c
```

Multi-threading is supported in special versions of all the Propeller 2 kernels – NMM, CMM and LMM (the multi-threading versions of these kernels are selected automatically when a program is linked with the **libthreads** library).

Each thread is simply a C function with a prototype that looks like a C `main` function - i.e.:

```
int function(int argc, char *argv[]);
```

When the thread is started, the `argc` and `argv` parameters can be provided, and when the thread terminates, it can return an int value. A **typedef** for a pointer to such a function is also provided:

```
typedef int (* _thread)(int argc, char *argv[]);
```

This **typedef** is used in the `_thread_init` function. It is defined in the header file `catalina_threads.h`

In addition to multiple threads executing on the same cog, Catalina also provides the ability to run C programs (including multi-threaded C programs) on multiple cogs. See the section on **Multi-Cog Support** for more details.

Fundamental Thread Functions

The fundamental thread library functions are defined in `catalina_threads.h`. They are as follows:

```
int _thread_get_lock();
```

Get the cog lock allocated to the kernel for context switching. See the explanation of `_thread_set_lock` for details on when this function is required.

```
void _thread_set_lock(int lock);
```

Set the cog lock the multi-threading kernel will use for context switching. If there are multiple multi-threading kernels started, it is important that they all use the same cog lock to prevent context switching contention.

Initially, each multi-threading kernel will use cog lock 7 – but the kernel does not reserve this lock via `_locknew`, so a new cog lock should usually be reserved using `_locknew` and then set using `_thread_lock_set` before the kernel starts any threads).

This can be done very simply by:

```
_thread_set_lock(_locknew());
```

Because the initial cog lock is not reserved, it does not need to be returned using `_lockret` – but if another lock is used and it subsequently needs to be changed, the following sequence *must* be used:

1. get the current cog lock via `_thread_get_lock`
2. reserve a new cog lock via `_locknew`
3. set the new cog lock in *all* multi-threading kernels via `_thread_set_lock`

4. release the current cog lock via `_lockret`

```
int _thread_ticks(void * thread_id, int ticks);
```

Update the tick count of the specified thread. Each tick is approximately 100 microseconds, and the thread will execute for this many ticks before a context switch (unless something occurs – such as a call to `_thread_yield()` – which makes the thread switch earlier.

A thread can update its own tick count, but the change will not take effect until the next context switch.

```
void * _thread_id();
```

Return the unique non-zero thread id of the current thread.

```
void * _thread_start(_thread PC, void * SP, int argc, char *argv);
```

Start a new thread. The SP must point to the top of at least `THREAD_BLOCK_SIZE` longs. These longs are used as the thread block. The RAM below this is then used as the stack of the thread.

Returns the (non-zero) thread id on success, or 0 on failure.

```
void * _thread_stop(void * thread_id);
```

Stop a thread executing.

Returns the non-zero thread id if the thread was found and stopped, or 0 if not.

```
void * _thread_join(void * thread_id, int * result);
```

Wait for a thread to complete and fetch its return value. Note that this function does not return until the thread has stopped.

Returns the non-zero thread id if the thread was found, or 0 if not. Also returns zero if you attempt to join your own thread, or the thread you are trying to join gets terminated.

```
void * _thread_check(void * thread_id);
```

Check if the specified thread is currently executing.

Returns the non-zero thread id if the thread is executing, or 0 if not.

```
void _thread_yield();
```

Yield the cog to another thread. This is typically called instead of "busy waiting" when a thread discovers it has no more work to do, and must wait for another thread, or for an external event.

```
int _thread_init_lock_pool (void * pool, int size, int lock);
```

Initialize a block of Hub RAM as a pool of locks. This function should be called once (and only once) for each pool. The pool must be (size + 5) bytes of Hub RAM, and must be long aligned.

If the initialization succeeds, 0 is returned.

```
int _thread_locknew(void * pool);
```

Allocate a free lock from a lock pool. Note that the pool must have previously been initialized using `_thread_init_lock_pool`. The id of the next unused lock in the pool is returned on success (1 .. size), and the lock is cleared.

If no more locks are available, -1 is returned.

```
int _thread_lockclr(void * pool, int lockid);
```

Clear the specified lock (1 .. size) in the specified lock pool. The lock pool must have been initialized using `_thread_init_lock_pool`, and the lock must have previously been allocated using `_thread_locknew` or an error is returned.

The previous value of the lock (0 or 1) is returned. On error, -1 is returned.

```
int _thread_lockret(void * pool, int lockid);
```

Return a lock (1 .. size) to the specified lock pool. The lock pool must have been initialized using `_thread_init_lock_pool`, and the lock must have previously been allocated using `_thread_locknew` or an error is returned.

On success, 0 is returned. On error, -1 is returned.

```
int _thread_lockset(void * pool, int lockid);
```

Set the specified lock (1 .. size) in the specified lock pool. The lock pool must have been initialized using `_thread_init_lock_pool`, and the lock must have previously been allocated using `_thread_locknew` or an error is returned.

The previous value of the lock (0 or 1) is returned. On error, -1 is returned. To check that the lock was not already set, test for a return value of 0.

```
int _thread_affinity(void *thread_id)
```

Return the affinity status of the specified thread (can be used to determine both the current affinity, and also the current state of any outstanding affinity request).

```
int _thread_affinity_stop(void *thread_id)
```

Stop the specified thread, which may have a different affinity from the calling thread.

Returns an error if an affinity command is already set for the specified thread.

```
int _thread_affinity_change(void *thread_id, int affinity)
```

Request a change of affinity for the specified thread. Check the affinity of the thread later to see if the change has taken effect.

Additional Thread Utility Functions

Some additional thread library functions are defined in *thread_utilities.h*. They are as follows:

```
int _thread_cog(_thread func, unsigned long *stack, int argc, char
               *argv[]);
```

Start a new multi-threaded kernel on a new cog, and have it initially run the specified thread.

```
int _thread_integer(int lock, int num);
```

This function is just the equivalent of the HMI function **t_integer**, but it uses a lock to ensure that only one such function can access the HMI plugin at once – this makes these functions more suitable for use when multiple threads are executing.

```
int _thread_unsigned(int lock, unsigned num);
```

As above, for **t_unsigned**.

```
int _thread_string(int lock, char *str);
```

As above, for **t_string**.

```
int _thread_char(int lock, char ch);
```

As above, for **t_char**.

```
int _thread_hex(int lock, unsigned num);
```

As above, for **t_hex**.

```
int _thread_bin(int lock, unsigned num);
```

As above, for **t_bin**.

```
int _thread_printf(int lock, char *str, ...);
```

As above, for **t_printf**.

```
void randomize();
```

While not specific to threads, this function is useful - it initializes the random number generator (using **srand**) based on the current clock value.

```
int random(int max);
```

While not specific to threads, this function is useful – it returns a random number from 0 to MAX - 1.

Read the README file in the *demo/multithread* directory for more details.

Multi-Model Support

Catalina allows the Propeller 1 to simultaneously execute LMM, CMM and XMM programs. Each program is executed on a separate cog and essentially runs independently. However, all programs share the same registry and all programs can use any of the loaded plugins.

There is only one program binary. When this binary is loaded and started, it starts the "primary" program, which must explicitly start each of the "secondary" programs, which are stored either in the memory space of the primary program (useful if you have XMM RAM) or on SD Card (useful if you do not have XMM RAM), and only loaded into Hub RAM when they are to be executed. For this reason, on the Propeller 1, Multi-Model support is generally most useful when the primary program is an XMM (SMALL or LARGE) program, otherwise it is wasteful of Hub RAM. In either case, the secondary programs will then use no Hub RAM until executed by the primary program.

Each secondary program can share a variable with the primary program – this variable can either be a simple variable or a structure, if a single variable is not sufficient. Each secondary program can use a different variable, or they can all share the same variable.

The compilation process for Multi-Model support requires that each secondary program be compiled first, as either a TINY (LMM) or COMPACT (CMM) program, with all the usual command-line options, plus the following additional options:

```
-R XXXXXX -C NO_ARGS -M64k
```

A binary file must be generated for each secondary program. However, this binary is not executed directly – it must be processed for inclusion either as an array in the primary program or as an overlay file that can be loaded off SD Card by the primary program. This is done using an updated version of the **spinc** utility.

The primary program that loads and executes the secondary programs (and runs in parallel with them) can be a TINY (LMM), COMPACT (CMM), SMALL (XMM) or LARGE (XMM) program.

The **XXXXXX** value represents the address at which the secondary program is to be loaded for execution, and must be determined according to the size of each secondary program, its run-time space requirements, and also by the necessity to not interfere with the reserved memory at the top of Hub RAM – i.e. the registry, cache, plugin data etc. This value may have to be established partly by trial and error, but it can also be done by simply reserving a suitable amount of Hub RAM as a local variable in the main function of the primary program, determining where the Hub RAM address of that local variable is, and then using that value for the -R parameter. An example of doing this is provided in the Multi-Memory Model demo programs.

The **-C NO_ARGS** is required to disable the usual C command-line argument processing in the secondary program (but not the primary program). Command-line arguments are not supported for secondary programs that are to be dynamically loaded, and it would also interfere with the passing of the parameter that specifies the shared variable address from the primary program to the secondary program.

The -M64k option may be required to allow CMM or LMM code to be compiled in areas of upper Hub RAM that would not normally be permitted for primary program code, but are permitted for secondary programs. Without this option, specifying a -R option in high Hub RAM can make the secondary program's binary size exceed the

usual 32k limit - which is not normally supported for LMM or CMM programs on the Propeller 1, but is for secondary programs that are to be dynamically loaded. There is no problem in simply always including this option when compiling programs intended to be dynamically loaded.

To run MULTIPLE subsidiaries simultaneously you must calculate a different value of **XXXXXX** for each secondary, and ensure they will not overlap when loaded. The runtime size of each secondary can be determined as described below.

Each secondary program must be compiled and then turned into a "blob" (i.e. a "binary large object", or "binary loadable object") for inclusion in the primary program. The blob can be an array in the include file generated by **spinc**, or as a separate binary file.

Producing an array blob:

Building a blob as an array in an include file is done using an updated version of the **spinc** utility, using a new command-line option (**-B**). For example:

```
spinc -B2 -c secondary_1.binary >blob_1.inc
spinc -B2 -c secondary_2.binary >blob_2.inc
(etc)
```

The new option (**-B**) specifies that a blob is to be created from one of the objects in the binary. It accepts a parameter to specify the object number, which (for Catalina Propeller 1 binaries) should generally be the SECOND object within the binary. For Catalina binaries the second object will be Catalina.spin – i.e. the object produced by the Catalina compiler from the C source code. So this option should (currently) always be specified as **-B2**.

The **-c** option specifies that a suitable 'start' function will be generated in the include file, suitable for starting the C program from the primary program. The name used for the function will be the name of the binary (i.e. "start_<<binary file name>>"). This can be overridden using the **-n** option if required (for instance, if you are generating multiple loadable programs from the same secondary binary). So, for example, if your binary file name is *hello_world.binary*, the start program generated would be:

```
int start_hello_world(void* var_addr, int cog)
```

This function accepts the address of a shared variable and a cog number on which the secondary program will be run (which can be the special constant **ANY_COG**). The function will return the actual cog number used, which can be used later to stop the secondary program.

If the object cannot be found or does not have the correct type (it must be an LMM or CMM object) then an error message will be issued. Additional runtime space to be allocated for the blob can be specified using the **-s** option. If no size is specified, 80 bytes (20 longs) is the default. For example, to instead specify run-time space of 200 bytes (50 longs):

```
spinc -B 2 -s 200 -n my_blob secondary.binary >blob.inc
```

As shown above, the **-n** option can be used to specify a name for the secondary, otherwise the binary file name is used. This name will be used for the start function, the blob itself, and all the constants defined for the blob. The output is a C source file, suitable for inclusion in the primary program.

The output of the **spinc** program is intended to be redirected to an include file that will be included in the primary program. Not only does it contain the start function and the blob in the form of an array, but also (as a series of **#defines**) the necessary addresses and sizes associated with the blob. This information can be used to allocate runtime space for the blob.

The **#defines** are all named in a similar way to the start function, by preceding them either with the binary file name, or the value of the **-n** parameter. The most important ones are:

```
#define <<name>>_BLOB_SIZE      0xFFFF // size of the blob in bytes
#define <<name>>_RUNTIME_SIZE  0xFFFF // runtime size in bytes
#define <<name>>_CODE_ADDRESS  0xFFFF // Hub RAM Address for execution
```

These and other constants are used by the start function, and can also be used in the primary program. For instance, to reserve runtime space for a secondary called **My_Prog**, and also to print it at runtime so that it can be used in the **spinc** process, the primary program might include code similar to the following:

```
#include "My_Prog.inc"

void main(void) {
    char RESERVED_SPACE[My_Prog_RUNTIME_SIZE];

    ...
    if ((int)&RESERVED_SPACE != My_Prog_CODE_ADDRESS) {
        printf("Recompile My_Prog using -R 0x%x\n",
            (int)&RESERVED_SPACE);
    }
    ...
}
```

The **spinc** process must be repeated for each secondary program (each one must have a unique name, even if the same binary is used to create them).

For examples of producing and using blobs, see the `demos\multimodel` directory.

Multi-Models and Multi-threading

The Multi-Model support works with programs that use multi-threading, but things can get a little complicated. XMM programs do not support multi-threading, but an XMM primary program can start non-XMM secondary programs that do.

First of all, note that if a primary or secondary program is multi-threaded, then ALL the dynamic kernels started by that program using the existing **_thread_cog()** function or the new **_threadstart_C()** functions will also be multi-threaded, and use the same memory model. However, threads cannot be shared between primary and secondary programs. So while each primary or secondary program can share

threads with any kernels they start using the `_thread_cog()` or `_threadstart_C()` functions, they cannot share threads with other primary or secondary kernels started with the new multi-model start functions, even if the programs are identical and use the same memory model. Each primary and secondary program (and any additional kernels they start) constitute a separate and isolated "world" as far as threads are concerned.

However, it is perfectly feasible for a threaded primary program to start a non-threaded secondary program, or vice versa. To facilitate this, additional threaded start functions have been added to explicitly start a secondary program as a multi-threaded program, corresponding to the various `_cogstart` functions:

`_threaded_cogstart_CMM_cog()` - start a blob as a threaded CMM program on a specific cog

`_threaded_cogstart_LMM_cog()` - start a blob as a threaded LMM program on a specific cog

Since the most likely case is that the primary and secondary programs will both be threaded, or both be non-threaded, by default the **spinc** utility will use the threaded or non-threaded start functions based on whether the primary program is threaded or non-threaded. However, if your primary program is threaded and your secondary program is not (or vice-versa) then you can override the default using one of two methods:

1. You can manually specify which start function to use using the -f command-line option to the **spinc** utility.

For example, to specify the non-threaded start function be used, even though the primary program is threaded:

```
catalina -lc -C NO_ARGS secondary.c
spinc -B2 -f _cogstart_LMM_cog -c secondary.binary
>secondary.inc
catalina -lc -lthreads primary.c
```

Or, to specify the threaded start function be used, even though the primary program is non-threaded:

```
catalina -lc -lthreads -C NO_ARGS secondary.c
spinc -B2 -f _threaded_cogstart_LMM_cog -c secondary.binary
>secondary.inc
catalina -lc primary.c
```

See the `build_all` script in the `demos\multimodel` directory, and the instructions for building `run_dining_philosophers.c` for an example of this method.

2. You can specify which start function to use in the program itself, by defining one of the following two symbols prior to including the output of the **spinc** utility:

```
#define COGSTART_THREADED /* use threaded start functions */
```

Or

```
#define COGSTART_NON_THREADED /* use non-threaded start
functions */
```

See the file *run_dining_philosophers.c* in the *demos\multimodel* directory for an example of this method. Note that the use of the first method (described above) overrides the use of this second method.

Multi-Models and Overlays

When using XMM modes, it makes perfect sense to create secondary programs in XMM RAM and use them as "overlays". The secondary programs occupy no valuable Hub RAM until they are needed. When they are needed they are loaded from XMM RAM into Hub RAM for execution, and they can be terminated and the same Hub RAM used for other purposes – such as loading another secondary program.

However, Catalina also allows programs to load overlays from files on an SD Card, which is useful on platforms without XMM RAM. To make use of this, the **spinc** program is used with a new option specified (-o 'name'), which will generate an output file (called 'name') containing the binary blob instead of an array. Using this option will also generate a start function that will load the blob from the named file at run time. For example:

For a non-threaded overlay:

```
catalina -lc -R 0x4000 -C NO_ARGS secondary.c
spinc -B2 secondary.binary -o blob.ovl >secondary.inc
catalina -lcx primary.c
```

For a threaded overlay:

```
catalina -lc -lthreads -R 0x4000 -C NO_ARGS secondary.c
spinc -B2 -f _threaded_cogstart_LMM_cog secondary.binary -o blob.ovl >secondary.inc
catalina -lcx -lthreads primary.c
```

Note that an include file is still generated, and must still be included in the primary program even though the blob itself is written to the overlay file. The include file contains the start function that will load the blob from the named file on the SD Card at run-time.

Of course, the primary program must be built with an SD Card file system to be able to load the overlays (in the cases above, this is accomplished by linking with the extended libraries via the -lcx command line option), and that the overlay files must exist on an SD Card which is inserted into the Propeller when the primary program is started (in the examples above, that file will be called 'blob.ovl').

An overlay file is not a normal Catalina binary - it is just the code and initialized data segments of the secondary program. The secondary program must be compiled to run at the correct address when loaded, and the start function will load the overlay file into this location in Hub RAM.

A simple example of overlays is included in the *demos\multimodel* directory.

Multi-processing, Locks and Memory Management

Beginning with release 5.0, Catalina now has more sophisticated support for multi-processing (i.e. multi-threading, multi-cog and multi-model) programs. In all these cases, it is critically important to protect and manage resources such as memory, and the services provided by plugins.

Catalina uses various techniques, described in the following sections.

Thread-safe Memory Management

First, and perhaps most importantly, in Catalina 5.0 the standard C memory management functions (**malloc**, **realloc**, **calloc**, **free**) and a new system break function (**sbrk**) are all thread safe. Since this incurs a small performance penalty, this is done by default only for programs that also use multi-threading, where it is most likely to be an issue – but in all cases you can decide to have these functions either made thread safe or not. Even in a multi-threaded program, if only one thread does memory allocation then there is no need for a lock and you can decide not to make these functions thread safe if you find it affects your program performance.

If your program already uses the thread library, you will probably not need to do anything to make your memory management thread safe. Such programs should already call the **_thread_set_lock()** function, which now also allocates a memory lock (unless one has already been allocated).

To specify a memory lock should be used in other situations (such as in multi-cog or multi-model programs where more than one program might use dynamic memory allocation), or to manually specify the lock to use, first allocate a lock using the normal **_locknew()** function, and then use the function **_memory_set_lock()** which tells the memory management functions to use the specified lock. Note that the lock only has to be set by one program even if there are programs running on multiple cogs and/or in multiple memory models – the lock itself is stored in a global variable and will be used by all the Catalina programs currently executing.

You can just retrieve the memory management lock that is currently in use without affecting anything by calling the function **_memory_get_lock()**. It will return -1 if no memory management lock is in use.

Note that to use a specific lock in a multi-threaded program, you must call **_memory_set_lock()** BEFORE calling **_thread_set_lock()**.

You can stop using the current memory lock by calling **_memory_set_lock()** with a value of -1, but note that this does not release the lock – to do that you can either use the value returned from this function (which will be the lock that was in use or -1 if none was in use) in a call to the **_lockret()** function.

If you use **_sbrk()** you can continue to do so, but it remains potentially thread unsafe. But there is now also a thread safe version called **sbrk()** (i.e. without the underscore) so if you use multi-threading you may want to modify your programs to use that function instead.

A demo program that tests the new thread safe malloc has been added to the `demos\multithread` folder – it is called `test_thread_malloc.c`.

Memory Fragmentation Management

In Catalina 5.0, the dynamic memory management has been modified to make it less prone to running out of memory when lots of small randomly sized chunks of memory are allocated, freed or re-allocated. This can lead to such extreme memory fragmentation that even though plenty of memory is available, none of the blocks are of useful size. To address this, some changes to when blocks are fragmented have been implemented (which in many cases is enough to fix the problem on its own), plus the internal memory defragmenter has been made manually callable as **`malloc_defragment()`**.

This means the defragmenter can be manually invoked periodically to recover from extreme memory fragmentation should it occur. Doing so is typically only required by programs that allocate many small randomly sized chunks of memory – programs that mostly allocate fixed block sizes (e.g. buffers of a standard size) will not be affected and don't need to manually run the defragmenter. The problem occurs mostly with multi-threaded programs because the stack requirements of such programs tends to be much larger, but there is no easy way to tell when heap memory is getting low, which is the only time the original memory management code called the defragmenter.

The configurable constant `MIN_SPLIT` has been introduced (defined in `impl.h` in the standard library sources). This is the smallest amount that must be left before a memory block will be split in two if it is used to allocate (or reallocate) a memory block smaller than its actual size – otherwise the block is left intact. This can significantly reduce memory fragmentation, which can lead to no memory blocks of sufficient size being available to satisfy a memory allocation request, even though there is plenty of memory free. `MIN_SPLIT` must be a power of 2. By default it is set to 64 (bytes).

The memory management constant `GRABSIZE` (the minimum size that malloc is configured to grab whenever it needs more RAM space) has also been set to `MIN_SPLIT`. In previous versions of Catalina it was set to 512, but this is too large for programs (like Lua) that do extensive allocation of small memory blocks. The previous value can be restored if required (edit `impl.h`) but the Catalina libraries will have to be recompiled.

Thread, Memory and Service Locks

In addition to the existing thread lock, and the new memory lock, Catalina 5.0 also provides a new function that can be used to assign one or more locks to protect the services offered by plugins in multi-processing programs. The **`_set_service_lock()`** function should be used when the same services may be invoked from multiple cogs or threads, which can lead to one service call being corrupted by another. You can elect to assign the same (specified) lock to protect all services, or assign a different lock per plugin.

The parameter to **_set_service lock()** can be a specific lock number to use (0.. 7 on the Propeller 1, or 0..15 on the Propeller 2) in which case the same lock will be used to protect all plugins, or -1, in which case a different lock will be assigned to each plugin. The latter option does the same job as defining the Catalina symbol **PROTECT_PLUGINS** on the command-line. If you do that, then calling **_set_service_lock()** does nothing, so it is safe to use both methods on the same program.

This means there are now *three* locking mechanisms provided by Catalina 5.0, intended for use by multi-processing programs. The only one that **MUST** be used is the one provided by **_set_thread_lock()** - this is *required* in multi-threaded programs because a thread lock is necessary for the multi-threading to work correctly. The others are optional, since they may affect performance and are only required when the program calls memory allocation or services from more than one execution stream (i.e. from more than one thread or cog).

The three lock functions are:

- _set_thread_lock()** *must* be called by multi-threaded programs. Sets both a thread lock and a memory lock (if a memory lock has not already been set).
- _set_memory_lock()** *can* be called by multi-cog or multi-model programs to set just a memory lock (multi-cog programs do not need to set a thread lock).
- _set_service_lock()** *can* be called by multi-model, multi-cog or multi-threaded programs to set one or more service locks.

In the case of a multi-threaded program, Catalina assumes that a memory lock and a service lock will both be required, so it allocates them automatically. But it is more complex in other cases – for instance, what looks like a single cog program may be executed using Catalina's multi-model support, which effectively makes the single cog program part of a multi-cog program AFTER compilation.

Note: Prior to release 5.0, Catalina used 4 bits for the lock in the service registry. The Propeller 1 only had 8 locks, so this was sufficient to represent all possible locks plus a bit to indicate "no lock". But the Propeller 2 has 16 locks, so Catalina must now use 5 bits for the lock. To make space for the extra bit, the number of bits used to hold the cog-specific service code has been reduced from 8 to 7. This still allows up to 127 services, but less than 40 such services are implemented even by the most complex plugins (the HMI plugins), so this is unlikely to be a problem. If it becomes so, the size of each service entry can simply be increased from a 16 bit word to a 32 bit long.

Although this change is only required for the Propeller 2, to keep the library functions consistent, it has been made on the Propeller 1 as well.

Posix threads (pthreads) support

Catalina supports a "thin" binding to Posix threads (aka pthreads). It is a "thin" binding because nearly all the Posix pthreads functions are simply mapped to the existing Catalina threads functions wherever possible. The pthreads functions are included in the existing Catalina libthreads library (i.e. you just link your program with **-lthreads** as normal). The include file *pthread.h* describes the pthread support in more detail. The support is quite comprehensive, except for a few of the more esoteric functions, and those things that are dependent on having an actual Posix operating system.

Note that when the pthread library is in use, Catalina sets the thread lock automatically for multi-threaded programs. This is done if no thread lock has been manually specified when the first pthread is created, by a call such as:

```
_set_thread_lock(_locknew())
```

However, this must still be done manually for multi-kernel programs (e.g. multi-cog or multi-model programs), since the same lock must be specified in *all* kernels.

Here is a brief summary of the level of support for Posix threads. For more information on Posix functions, refer to any standard Posix documentation. For example, a comprehensive introduction can be found at:

<https://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/91-pthreads/Lib/pthreads-book.pdf>

Posix pthread functions supported

The following functions are implemented using Catalina threads:

Spinlock functions:

- pthread_spin_init
- pthread_spin_destroy
- pthread_spin_lock
- pthread_spin_unlock

Mutex attribute functions:

- pthread_mutexattr_init
- pthread_mutexattr_destroy
- pthread_mutexattr_settype
- pthread_mutexattr_gettype

Mutex functions:

- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutex_lock
- pthread_mutex_timedlock

Read/write lock attribute functions:

- pthread_rwlockattr_init

- `pthread_rwlockattr_destroy`

Read/write lock functions:

- `pthread_rwlock_init`
- `pthread_rwlock_destroy`
- `pthread_rwlock_rdlock`
- `pthread_rwlock_trywrlock`
- `pthread_rwlock_unlock`
- `pthread_rwlock_timedrdlock`
- `pthread_rwlock_timedwrlock`

Barrier attribute functions:

- `pthread_barrierattr_init`
- `pthread_barrierattr_destroy`

Barrier functions:

- `pthread_barrier_init`
- `pthread_barrier_destroy`
- `pthread_barrier_wait`

Condition variable attribute functions:

- `pthread_condattr_init`
- `pthread_condattr_setclock`
- `pthread_condattr_getclock`

Condition variable functions:

- `pthread_cond_init`
- `pthread_cond_broadcast`
- `pthread_cond_destroy`
- `pthread_cond_timedwait`

Posix pthread attribute functions:

- `pthread_attr_init`
- `pthread_attr_destroy`
- `pthread_attr_getstack`
- `pthread_attr_getstacksize`
- `pthread_attr_setguardsize`
- `pthread_attr_getguardsize`
- `pthread_attr_getdetachstate`
- `pthread_attr_setdetachstate`
- `pthread_attr_setscope`
- `pthread_attr_getscope`
- `pthread_attr_setschedparam`
- `pthread_attr_getschedparam`
- `pthread_attr_setschedpolicy`
- `pthread_attr_getschedpolicy`

Posix pthread functions:

- `pthread_create`
- `pthread_cancel`
- `pthread_join`
- `pthread_once`
- `pthread_detach`
- `pthread_equal`
- `pthread_exit`
- `pthread_self`

Posix clock functions:

- `pthread_getcpuclockid`
- `pthread_setschedparam`
- `pthread_getschedparam`
- `pthread_setcancelstate`

The following Posix functions are defined, but for compliance only. They may either do nothing (because nothing is required for Catalina's implementation) or just return values appropriate to Catalina's implementation:

- `pthread_mutex_consistent`
- `pthread_barrierattr_getpshared`
- `pthread_barrierattr_setpshared`
- `pthread_mutexattr_setpshared`
- `pthread_mutexattr_getpshared`
- `pthread_mutexattr_getrobust`
- `pthread_mutexattr_setrobust`
- `pthread_rwlockattr_getpshared`
- `pthread_rwlockattr_getpshared`
- `pthread_setconcurrency`
- `pthread_getconcurrency`
- `pthread_condattr_destroy`
- `pthread_condattr_setpshared`
- `pthread_condattr_getpshared`

Posix pthread functions not supported

The following pthreads functions are neither defined or implemented, typically because a program that uses these probably won't work correctly without them, or just by using default values for the relevant attributes – the reason for the use of these functions should be investigated:

- `pthread_atfork`
- `pthread_testcancel`
- `pthread_setschedprio`
- `pthread_cleanup_push`
- `pthread_cleanup_pop`
- `pthread_setspecific`
- `pthread_getspecific`
- `pthread_key_create`

- `pthread_key_delete`
- `pthread_attr_setinheritsched`
- `pthread_attr_getinheritsched`
- `pthread_mutex_setprioceiling`
- `pthread_mutex_getprioceiling`
- `pthread_mutexattr_setprioceiling`
- `pthread_mutexattr_getprioceiling`
- `pthread_mutexattr_setprotocol`
- `pthread_mutexattr_getprotocol`

Non-Posix pthread functions added

The following functions have been added because they are generally useful, or because they provide Propeller-specific functionality. See the file *pthreads.h* for more details:

- `pthread_yield`
- `pthread_sleep`
- `pthread_msleep`
- `pthread_usleep`
- `pthread_printf`
- `pthread_createaffinity`
- `pthread_setaffinity`
- `pthread_getaffinity`

Random Number Support

Pseudo Random Numbers

Catalina supports the standard C functions **rand()** to return a pseudo-random number, and **srand()** to seed the pseudo-random number generator.

Catalina adds a **getrand()** function (defined in *propeller.h*) which is implemented on both the Propeller 1 and the Propeller 2 and which returns 32 bits of pseudo random data. A program that demonstrates the use of the function has been added in *demos\examples\ex_random.c*

The first time this function is called it calls **srand()** with the current system counter value and is therefore best called after some user input or other random source of delay. It then returns the result of 3 combined calls to **rand()** to make up 32 random bits (rand itself only returns 15 bits).

This means you can either use just this function, or use this function once to generate a seed for **srand()** and thereafter use **rand()**, which is what most traditional C programs would typically do.

Note that **rand()** only returns a value between **0** and **RAND_MAX** (inclusive) (i.e. 0 .. 32727 on the Propeller) whereas **getrand()** returns 32 bits.

To simulate **rand()** using **getrand()**, use an expression like:

```
(getrand() % (RAND_MAX + 1))
```

True Random Numbers

The Propeller 2 has an internal random number generator, which is accessible via **getrealrand()** function, so Catalina also adds a similar function for use on the Propeller 1. To support this, a random number plugin has been added for the Propeller 1, based on Chip Gracey's RealRandom.spin. The plugin is enabled by defining the Catalina symbol **RANDOM**. For example:

```
catalina ex_random.c -lci -C RANDOM
```

The RANDOM plugin occupies a cog on the Propeller 1. It generates random data continuously, and writes a new 32 bit random number to the second long in its registry entry approximately every 100us (at 80Mhz).

The **getrealrand()** function will generate unique random numbers in situations where the Propeller 1 previously would not, such as if a program was automatically started immediately after power up. In such situations the system clock would be reset, so the previous technique of seeding a pseudo random number generator with the system clock would always generate the same sequence of pseudo random numbers.

Random Number Functions

Here is a summary of the random number routines:

rand(), srand()	rand() generates pseudo random numbers, in the range 0 to MAX RAND (32767), and srand() can be used to seed the random number generator
getrand()	getrand() generates 32 bits of pseudo random data using rand() . On first call it also seeds the random number generator using srand() and the current system clock. This avoids the need to explicitly call srand()
getrealrand()	getrealrand() returns 32 bits of random data. On the Propeller 1 this will be true random data if the RANDOM plugin is loaded, otherwise it will use the same technique as getrand() . On the Propeller 2 it always returns true random data.

The program *ex_random.c* in *demos/examples* demonstrates both **rand()** and **getrand()** (which both generate pseudo random numbers) as well as the **getrealrand()** function (which generates true random numbers).

Plugin Support

Catalina was designed to be open and extendable. It provides a standard interface to PASM programs running in a cog by defining a common “plugin” interface to these programs.

A “plugin” is just a SPIN object that contains a PASM program which runs on a cog – and which is registered with the Catalina Kernel. Once registered, the functions provided by the plugin can be invoked from within a Catalina C program.

A plugin is typically a device driver, but is not limited to that. Catalina uses plugins internally in various ways. For example:

- All keyboard/mouse/screen access is via various **HMI** plugins. These plugins act as wrappers for other drivers. The wrapper not only provides a uniform means of accessing different Parallax drivers, it adds many functions that do not exist in the underlying drivers – such as screen scrolling and cursor support;
- The Float32Full and Float32A functions are accessed by turning the standard Float32A and Float32Full cog programs into the plugins **Float32_A** and **Float32_B**.
- Real-Time Clock support is implemented as a **CLOCK** plugin.
- Gamepad support is implemented as a **GAMEPAD** plugin.

Existing cog programs can often be used “as is” in conjunction with Catalina if no access is required to them from the Catalina program – just add the appropriate SPIN objects into a new customized target (see the **Customized Targets** section later in this document) and then build the Catalina program for that target.

Cog functions

Catalina provides C functions that mimic the Parallax operations used for managing cogs, interacting with locks, waiting for various conditions, or interacting with the special cog registers. Catalina also provides *direct access* to the Propeller special registers, which is described in the next section (and which may be more familiar to existing Spin programmers).

The following functions are defined in the include file **catalina_cog.h**:

```
unsigned _clockfreq();
```

This function returns the current clock frequency, as found in the long at hub RAM address 0.

```
unsigned _clockmode();
```

This function returns the current clock mode, as found in the byte at hub RAM address 4. To assist in decoding the returned values, see the symbols defined in the include file.

```
unsigned _clockinit(unsigned mode, unsigned freq);
```

This function can be used to set both the current clock mode and frequency. To assist in specifying clock mode values, see the symbols defined in the include file.

```
int _cogid();
```

This function returns the cog id (0 .. 7) of the cog in which this C program is executing.

```
int _coginit(int par, int addr, int cogid);
```

This function starts an arbitrary PASM program in a new cog. The **par** and **addr** parameters must be given as **long** addresses (which can easily be done by dividing the normal **byte** addresses by 4). The **cogid** parameter can be a specific cog, or the special value **ANY_COG**. The **addr** parameter must be in Hub RAM. The program will be started using a non-threaded kernel in any cog. The actual cog number used will be returned by the function, or -1 on any error.

```
int _coginit_C(void func(void), unsigned long *stack);
```

This function starts a C void function that accepts no arguments in a new cog. The **stack** parameter must point to **the end of** an array of longs that will be used for the function's program stack. The C function and the stack must be in Hub RAM. The C function will be started using a non-threaded kernel in any spare cog. The actual cog number used will be returned by the function, or -1 on any error.

```
int _coginit_C_cog(void func(void), unsigned long *stack, unsigned int cog);
```

As above, except the program will be started in the specified cog.

```
int _cogstart_C(void func(void *), void *arg, void *stack, uint32_t size);
```

This function starts a C void function that accepts a void * argument in a new cog. The **stack** parameter must point to **the start of** an array of **size** bytes that will be used for the function's program stack. The **arg** will be passed to the new function on startup. The C function and the stack must be in Hub RAM. The C function will be started using a non-threaded kernel in any spare cog. The actual cog number used will be returned by the function, or -1 on any error.

```
int _cogstart_C_cog(void func(void *), void *arg, void *stack, uint32_t size, unsigned int cog);
```

As above, except the program will be started in the specified cog.

```
int _cogstart_CMM(uint32_t PC, uint32_t SP, void *arg);
```

This function starts a blob that contains a CMM program in a new cog. The C program can accept a void * argument. The Program must be in Hub RAM, and will be started at the address represented by **PC**. The **SP** parameter must point to the initial stack for the program (i.e. **the end of** the space reserved for stack), which must be large enough to accommodate the program's runtime needs. The **arg** will be passed to the new function on startup. The program will be started using a non-threaded kernel in any spare cog. The actual cog number used will be returned by the function, or -1 on any error.


```
int _cogstart_CMM_cog(uint32_t PC, uint32_t SP, void *arg, unsigned
int cog);
```

As above, except the program will be started in the specified cog.

```
int _cogstart_LMM(uint32_t PC, uint32_t SP, void *arg);
```

This function starts a blob that contains an LMM program in a new cog. The C program can accept a void * argument. The Program must be in Hub RAM, and will be started at the address represented by **PC**. The **SP** parameter must point to the initial stack for the program (i.e. **the end of** the space reserved for stack), which must be large enough to accommodate the program's runtime needs. The **arg** will be passed to the new function on startup. The program will be started using a non-threaded kernel in any spare cog. The actual cog number used will be returned by the function, or -1 on any error.

```
int _cogstart_LMM_cog(uint32_t PC, uint32_t SP, void *arg, unsigned
int cog);
```

As above, except the program will be started in the specified cog.

```
int _threaded_cogstart_CMM_cog(uint32_t PC, uint32_t SP, void *arg,
unsigned int cog);
```

This function starts a blob that contains a CMM program in a new cog. The C program can accept a void * argument. The Program must be in Hub RAM, and will be started at the address represented by **PC**. The **SP** parameter must point to the initial stack for the program (i.e. **the end of** the space reserved for stack), which must be large enough to accommodate the program's runtime needs. The **arg** will be passed to the new function on startup. The program will be started using a threaded kernel in the specified cog. The actual cog number used will be returned by the function, or -1 on any error.

```
int _threaded_cogstart_LMM_cog(uint32_t PC, uint32_t SP, void *arg,
unsigned int cog);
```

This function starts a blob that contains an LMM program in a new cog. The C program can accept a void * argument. The Program must be in Hub RAM, and will be started at the address represented by **PC**. The **SP** parameter must point to the initial stack for the program (i.e. **the end of** the space reserved for stack), which must be large enough to accommodate the program's runtime needs. The **arg** will be passed to the new function on startup. The program will be started using a threaded kernel in the specified cog. The actual cog number used will be returned by the function, or -1 on any error.

```
int _coginit_Spin(void *code, void *data, void *stack, int start, int
offs);
```

This function starts a Spin program in a new cog. The **code**, **data**, and **stack** parameters point to arrays containing the compiled Spin object code, the Spin programs var segment, and sufficient stack space to

execute the Spin program. The content of these arrays, and the **start** and **offs** parameters are typically populated using the output of the Catalina **spinc** utility when it is invoked with the **-c** flag. The code, data and stack arrays must be in Hub RAM. The Spin program will be executed in a new cog.

```
int _cogstop(int cogid);
```

This function stops the specified cog.

```
int _locknew();
```

This function checks out and returns the next available lock (0 .. 7), or returns -1 if no locks are available.

```
int _lockclr(int lockid);
```

This function clears the lock, and returns the previous value of the lock.

```
int _lockret(int lockid);
```

This function returns the specified lock to the pool of available locks.

```
int _lockset(int lockid);
```

This function sets the lock, and returns 1 if locking was successful, or 0 if not. Note that the Propeller 1 and Propeller 2 lock semantics are slightly different. On the Propeller 1 the lock is always locked, and success indicates that it was not already locked, whereas on the Propeller 2, the lock is only locked if the call is successful.

```
int _waitcnt(unsigned count);
```

This function performs a **waitcnt** instruction, waiting for the system counter to reach the specified count.

```
int _waitvid(unsigned colors, unsigned pixels);
```

This function performs a **waitvid** instruction, sending the specified data to the video circuitry. Note that while this is supported, it is very unlikely that a C program could execute waitvid instructions fast enough to implement a video driver. However, the waitvid instruction is sometimes used for other purposes.

```
int _waitpeq(unsigned mask, unsigned result, int a_or_b);
```

This function executes a **waitpeq** instruction, waiting for the specified register (a or b) to not equal the specified result. Use the values **INA** or **INB** to specify the register.

```
int _waitpne(unsigned mask, unsigned result, int a_or_b);
```

This function executes a **waitpne** instruction, waiting for the specified register (a or b) to not equal the specified result. Use the values **INA** or **INB** to specify the register.

```
unsigned _cnt()
```

This function returns the current value of the system counter.

```
unsigned _ina()
```

This function returns the current value of the **INA** register.

```
unsigned _inb()
```

This function returns the current value of the **INB** register.

```
unsigned _dira(unsigned mask, unsigned direction);
```

This function sets the current value of the **DIRA** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **DIRA**, so to determine its current value without changing it, specify both a mask and direction of zero.

```
unsigned _dirb(unsigned mask, unsigned direction);
```

This function sets the current value of the **DIRB** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **DIRB**, so to determine its current value without changing it, specify both a mask and direction of zero.

```
unsigned _outa(unsigned mask, unsigned output);
```

This function sets the current value of the **OUTA** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **OUTA**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _outb(unsigned mask, unsigned output);
```

This function sets the current value of the **OUTB** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **OUTB**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _ctr_a(unsigned mask, unsigned control);
```

This function sets the current value of the **CTRA** register. The mask can be used to specify the bits in the register that will be affected. There are definitions and macros to help set the values of the counter control bits in the file **catalina_cog.h**

The function returns the original value of **CTRA**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _ctrb(unsigned mask, unsigned control);
```

This function sets the current value of the **CTRB** register. The mask can be used to specify the bits in the register that will be affected.

There are definitions and macros to help set the values of the the counter control bits in the file **catalina_cog.h**

The function returns the original value of **CTRB**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _frqa(unsigned mask, unsigned frequency);
```

This function sets the current value of the **FRQA** register. The mask can be used to specify the bits in the register that will be affected. The meaning of the frequency bits depends on the setting of the **CTRA** register.

The function returns the original value of **FRQA**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _frqb(unsigned mask, unsigned frequency);
```

This function sets the current value of the **FRQB** register. The mask can be used to specify the bits in the register that will be affected. The meaning of the frequency bits depends on the setting of the **CTRB** register.

The function returns the original value of **FRQB**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _phsa(unsigned mask, unsigned phase);
```

This function sets the current value of the **PHSA** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **PHSA**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _phsb(unsigned mask, unsigned phase);
```

This function sets the current value of the **PHSB** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **PHSB**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _vcfg(unsigned mask, unsigned config);
```

This function sets the current value of the **VCFG** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **VCFG**, so to determine its current value without changing it, specify both a mask and output of zero.

```
unsigned _vscl(unsigned mask, unsigned scale);
```

This function sets the current value of the **VSCL** register. The mask can be used to specify the bits in the register that will be affected.

The function returns the original value of **VSCL**, so to determine its current value without changing it, specify both a mask and output of zero.

Catalina also provides two macros that can simplify the use of locks. Once a lock has been allocated – e.g. via a statement such as `lock = _locknew()` then the following macros can be used:

```
ACQUIRE(lock)
```

This macro causes the program to loop until it successfully acquires the specified lock.

```
RELEASE(lock)
```

This macro causes the program to release the specified lock.

Note that for users who intend porting code between Catalina and other Propeller C compilers, there is a header file called **catalina_icc.h** which “wraps” the Catalina specific cog function syntax within macros that can be easily redefined. This allows portable C code to be written. This file currently supports the Catalina and the ICC compilers, and may support other future compilers – see the file for more details.

propeller.h and Special Register Access

Some of the functions described in the previous section provide access to the Propeller's special registers (**INA**, **DIRA** etc). However, *direct access* is also provided simply by declaring the register names as **extern volatile**. This can be conveniently done by include the file **propeller.h**, which contains (among other things) the following definitions:

```
extern volatile const unsigned PAR;
extern volatile const unsigned CNT;
extern volatile const unsigned INA;
extern volatile const unsigned INB;
extern volatile unsigned OUTA;
extern volatile unsigned OUTB;
extern volatile unsigned DIRA;
extern volatile unsigned DIRB;
extern volatile unsigned CTRA;
extern volatile unsigned CTRB;
extern volatile unsigned FRQA;
extern volatile unsigned FRQB;
extern volatile unsigned PHSA;
extern volatile unsigned PHSB;
extern volatile unsigned VCFG;
extern volatile unsigned VSCL;
```

Once these names are declared as shown above (or **propeller.h** is included) the register names can be used like any other C variable in any C expression, without

being further defined, and they will represent the appropriate special propeller register. For example:

```
DIRA = 0xff000000 | INA;
OUTA |= 1
if (CNT == 0) ...
while ((INA & 0x00100000) == 0) ...
... etc ...
```

Note that if the special register names are **not** declared as **external volatile**, the names can be used as normal C variable names (of course, they will need to be declared the same way as any other C variable).

The include **propeller.h** also provides the following definitions which defines various macros designed to emulate the equivalent functions found in Spin:

COGID	return the cog number
COGSTOP(cog)	stop the specified cog
COGINIT(val)	start the cog with
LOCKNEW	allocate a lock
LOCKCLR(lock)	clear a lock
LOCKSET(lock)	set a lock
LOCKRET(lock)	release a lock
WAITCNT(count, ticks)	wait for cnt to equal count, then add ticks to count
WAITVID(colors, pixels)	execute WAITVID (Propeller 1 only)
WAITPNE(mask, pins)	execute WAITPNE (Propeller 1 only)
WAITPEQ(mask, pins)	execute WAITPEQ (Propeller 1 only)
CLKFREQ	return the Clock Frequency
CLKMODE	return the Clock Mode
CLKSET(mode, frequency)	set the Clock Mode and Frequency

The include file **propeller.h** file also defines some useful pin functions:

setpin(pin, value)	sets pin to output and sets value
setpin(pin)	sets pin to input and gets value
togglepin(pin)	sets pin to input and toggles value

Finally, **propeller.h** also defines the following convenient macros:

WAIT(ticks)	wait for a number of clock ticks
msleep(millisecs)	wait for a number of milliseconds
sleep(seconds)	wait for a number of seconds

Registry, Plugin and Service functions

Catalina provides functions for interacting with the registry that is used to record which plugins are currently loaded, to provide default communications blocks for each plugin, and to invoke the functions implemented by those plugins.

These functions are defined in the include file **catalina_plugin.h**. The functions are divided into three logical layers:

- Layer 1 – basic registry setup
- Layer 2 – plugin-based requests
- Layer 3 – service-based requests

The following Layer 1 functions are provided:

```
unsigned _registry();
```

This function returns the address of the registry. This is required to be passed to a cog when starting a dynamic kernel to execute C code on that cog.

```
void _register_plugin(int cog_id, int plugin_type);
```

This function can be used to register that a plugin of a specified type is running on a particular cog. Plugins must be registered before requests can be sent to them.

```
void _unregister_plugin(int cog_id);
```

This function can be used to unregister a plugin.

Plugin types 0 to 127 are reserved for various basic Catalina plugins (see the file **catalina_plugin.h** for a complete list of those currently allocated), but plugin types 128 to 254 are free for users to define for their own purposes.

The following Layer 1 macros are provided to simplify access to the registry:

```
REGISTRY_ENTRY(c)
```

This macro returns the registry entry for cog c. The parameter should be from 0 and 7 – any other value will return an undefined result. The result is an unsigned value.

```
REGISTERED_TYPE(c)
```

This macro returns the registered type for cog c. The parameter should be from 0 and 7 – any other value will return an undefined result. The result is an unsigned value.

```
REQUEST_BLOCK(c)
```

This macro returns a pointer to the request block reserved for cog c. The parameter should be from 0 and 7 – any other value will return an undefined result. The request block structure pointed to is defined as:

```
typedef struct {
    unsigned int request;
    unsigned int response;
} request_t;
```

The following Layer 2 functions are provided:

```
int _locate_plugin(int plugin_type);
```

This function can be used to find a cog on which a plugin type is executing. Note that if there is more than one plugin of a specified type executing, only the first will be found.

```
int _short_plugin_request (long plugin_type, long code, long param);
```

This function can be used to send a “short” request to a plugin specified by type (e.g. **LMM_HMI**). See the include file for a list of

plugin types. Short requests have a code and up to 24 bits of parameter. Note that the meaning of the code and the parameters is plugin-dependent, and also that different plugin types may require short or long requests to be used for specific request codes.

```
int _long_plugin_request (long plugin_type, long code, long param);
```

This function can be used to send a “long” request to a plugin specified by type (e.g. **LMM_HMI**). See the include file for a list of plugin types. This type of long request has a code and **one** 32 bit parameter. Note that the meaning of the code and the parameter is plugin-dependent, and also that different plugin types may require short or long requests to be used for specific request codes.

```
int _long_plugin_request_2 (long plugin_type,
                           long code,
                           long par1,
                           long par2);
```

This function can be used to send a “long” request to a plugin specified by type (e.g. **LMM_HMI**). See the include file for a list of plugin types. This type of long request has a code and **two** 32 bit parameters. Note that the meaning of the code and the parameters is plugin-dependent, and also that different plugin types may require short or long requests to be used for specific request codes.

```
float _float_request(long plugin_type, long code, float a, float b);
```

This function can be used to send a “long” request to a plugin specified by type (e.g. **LMM_HMI**). See the include file for a list of plugin types. This type of long request has a code and **two** 32 bit **floating point** values as parameters. Note that the meaning of the code and the parameters is plugin-dependent, and also that different plugin types may require short or long requests to be used for specific request codes.

The following Layer 3 functions are provided:

```
int _short_service (long svc, long param);
```

This function can be used to send a “short” request for a specific service (e.g. **SVC_T_CHAR**). See the include file for a list of services. Short requests have a code and up to 24 bits of parameter. Note that the meaning of the parameter is service-dependent, and also that different services may require short or long requests to be used.

```
int _long_service (long svc, long param);
```

This function can be used to send a “long” request for a specific service (e.g. **SVC_RTC_SETFREQ**). See the include file for a list of services. Long requests have a code and up to 32 bits of parameter. Note that the meaning of the parameter is service-dependent, and also that different services may require short or long requests to be used.


```
int _long_service_2 (long svc, long par1, long par2);
```

This function can be used to send a “long” request for a specific service (e.g. **SVC_SD_READ**). See the include file for a list of services. This type of long request has a code and **two** 32 bit parameters. Note that the meaning of the parameters is service-dependent.

```
float _float_service(long svc, float a, float b);
```

This function can be used to send a “long” request for a specific service by type (e.g. **SVC_FLOAT_ADD**). See the include file for a list of plugin types. This type of long request has a code and **two** 32 bit **floating point** values as parameters. Note that the meaning of the parameters is service-dependent.

The following Layer 3 macros are provided to simplify access to the service registry:

```
SERVICE_ENTRY(s)
```

This macro returns the registry entry for service **s**. The parameter should be from 1 to SVC_MAX – any other value will return an undefined result. The result is an unsigned short value.

```
SERVICE_COG(s)
```

This macro returns the cog containing the plugin which implements service **s**. The parameter should be from 1 to SVC_MAX – any other value will return an undefined result. The result is an unsigned value. A value of 0xF indicates the service is not currently implemented by any loaded plugin.

```
SERVIC_LOCK(s)
```

This macro returns the lock that must be successfully set to gain access to service **s**. The parameter should be from 1 to SVC_MAX – any other value will return an undefined result. The result is an unsigned value. A value of 0xF indicates the service is not currently implemented by any loaded plugin.

```
SERVICE_CODE(s)
```

This macro returns the request that will be sent to the plugin to request service **s**. The parameter should be from 1 to SVC_MAX – any other value will return an undefined result. The result is an unsigned value. A value of 0x00 indicates the service is not currently implemented by any loaded plugin.

The following miscellaneous utility function are provided:

```
char *_plugin_name(int type)
```

This function returns a pointer to a human-readable name for the plugin type. For example “Real-Time Clock” or “Gamepad”.

Note that the same basic plugin functions can generally be requested using either layer 2 or layer 3 requests. The advantage of using layer 3 requests is that layer 3

implements contention control (necessary if you have multiple threads or multiple cogs executing C programs), that you do not need to know the plugin type to request a service (i.e. allowing the same service to be implemented by different plugins in different targets), and also that layer 3 access is slightly faster.

Services 1 to 64 are predefined to mean various basic Catalina services (see the file **catalina_plugin.h** for a complete list), but services 65 .. 96 are free for users to define for their own purposes.

AN IMPORTANT NOTE ABOUT REGISTRY ACCESS: When accessing the registry, any addresses used in the registry, or in a plugin or service request, ***must be HUB RAM addresses***. This is because plugins are normally implemented as Spin/PASM programs that have ***no access to XMM RAM***. For example, if a service requires a parameter that represents an address where the plugin expects to find data to process, the address ***must be in Hub RAM***. If the data is actually located in XMM RAM, it must be copied to Hub RAM before the service request.

Debugger Support

Catalina now provides support for two different debuggers:

- **BlackBox** – a source level debugger with a command-line interface. BlackBox runs under both Linux or Windows, and is included with Catalina.
- **POD** – an assembly language debugger. Runs under both Linux or Windows

Note that POD is an assembly level debugger, while the others are source level debuggers. POD can be used to disassemble programs, and also examine the execution of programs within the Catalina kernel, while the others can only be used to view the C code being executed. While it is technically possible to use POD in conjunction with one of the other debuggers, this is not recommended since each debugger consumes a cog and also some RAM space – which means there wouldn't be much left for the actual program!

BlackBox Support

BlackBox support is enabled using the **-g** (or **-g3**) command-line option, or including that option in Geany.

For information about BlackBox, see the document **BlackBox Reference Manual**, and the tutorial document **Getting Started with BlackBox**.

For historical reasons, such as in some parts of the code, the debugger is referred to as BlackCat rather than BlackBox. **BlackCat** was a separate debugger developed jointly by Bob Anderson and Ross Higson which used the same protocol as BlackBox, but is no longer supported. Use BlackBox instead.

POD Support

Catalina comes with special debug targets intended to be used with the Propeller On-chip Debugger (**POD**). For information about **POD**, see the Parallax discussion forums – e.g. <http://forums.parallax.com/showthread.php?92924-UPDATED-Propeller-On-chip-Debugger>).

Compiling programs for debugging with POD

To build a debugging version of your program, simply use the pod target. For example:

```
catalina file.c -t pod
```

The debug targets add the **POD** runtime components to the Catalina Kernel. To access **POD**, the **PropTerminal** program must be used. **PropTerminal** can be downloaded from <http://insonix.ch/propeller/>. Binary files produced by Catalina can be loaded directly from the **PropTerminal** program. Set the size of the screen to 30 rows by 40 columns (by editing the *PropTerminal.ini* file).

PropTerminal is a Windows executable, but can be used under Linux with the **Wine** Windows emulator (see <http://www.winehq.org/>). However, configuring Wine under Linux is very system-dependent, and is beyond the scope of this document

Some limitations apply when using **POD**, which is quite resource hungry:

- **POD** requires some space in the Catalina Kernel, so the **debug** target makes space by moving the basic floating point operations out to the Float32_A plugin – this means floating point programs will execute somewhat slower when using **POD**.
- **POD** requires a cog, so the standard **debug** target loads the Fload32A plugin. If floating point support is not required, use a target that does not load any floating point, or use the target configuration options to disable floating point.
- **POD** is a SPIN program, and requires quite a lot of Hub RAM (about 12 kb) so large Catalina programs may have to be broken into parts when debugging.
- **POD** cannot be used to debug Catalina programs that use *malloc* – this is because the Catalina implementation of *malloc* assumes it will have access to all RAM space not used by Catalina itself – and this includes the VAR space used by **POD**.
- **POD** cannot be used with XMM programs – use **BlackBox** instead. See the section **XMM Support** for more details on XMM programs, or the document **Getting Started with BlackBox** or the **BlackBox Reference Manual**.

Since it has to be embedded within the Catalina Kernel, a special version of **POD** is included – this is a customized version which is “aware” of the Catalina Kernel, the special Kernel registers, and also has support for LMM programs. This can simplify debugging Catalina programs.

POD is automatically included with a Catalina program when one of the debug targets is specified (e.g. using the option `-t pod` to Catalina). Note that **POD** is not a source level debugger, so when debugging a Catalina program, it will usually be necessary to produce a listing of the actual LMM PASM code (which can be generated by using the Catalina command-line option `-y` ¹⁶

The debug targets use a special version of the Catalina Kernel that makes space for **POD** by moving the “native” support for the basic floating point operations (normally included in the Kernel) out to be handled by a plugin. This means that when debugging programs that use **ANY** floating point operations (not just the math library functions) your Catalina programs **MUST** be compiled with the `-lma` option to Catalina rather than the `-lm` option.

To save space (and also because the program being debugged often needs the HMI devices itself) all the pod target assumes the use of serial communications for all

¹⁶ Note that the addresses used in POD are the actual addresses, which may not match the addresses given in the listing – the listing shows address in the binary image, which is not necessarily the same as the address at which the Catalina program will load and execute.

communications between the user and **POD**. This is why it is necessary to use an external serial application (i.e. **PropTerminal**) when debugging.

Using POD

This document is not intended to be either a **POD** tutorial or a debugging tutorial. It only describes the features that have been added to **POD** specifically to support Catalina LMM PASM programs. These are:

- When displaying memory, CTRL+PAGEUP and CTRL+PGEDOWN have been implemented to move through memory a section at a time. When displaying hub RAM as longs, a section is 1024 longs, (or 4096 bytes). When displaying cog RAM, or hub RAM as bytes, a section is 64 longs (256 bytes).
- When displaying an assembly view, CTRL+L is used to toggle POD between LMM mode and COG mode. The two modes are as follows:

COG mode : In COG mode **POD** displays the cog RAM, with individual PASM instructions disassembled. You can set breakpoints and/or single step through individual kernel operations, including the fetching of each LMM instruction from hub RAM. The only difference between this mode and standard POD is that the LMM entry points in the kernel and the LMM registers are labeled (this is for convenience only, since the disassembled PASM code does not use these labels). For example, you may see code like:

```
000 Z    mov    $001, #001
001      rdlong $002, $001 WZ
002 NZ    add    $003, $002
```

LMM mode : In LMM mode POD displays the hub RAM, with individual Catalina LMM PASM instructions disassembled. You can set breakpoints and/or single step through individual LMM functions, including the special LMM instructions. Instructions that represent LMM instructions are shown using their LMM names. Parameters to these instructions are shown as simply 'long <value>'. Each LMM instruction is disassembled to show the LMM registers it uses (i.e. r0 ... r23 and the special registers PC, SP, FP, RI, BC, BA, BZ). For example, you may see code like:

```
0000      NEWF
0004      sub    SP, #4
000C      LODL
0010      long   $00001000
0014      adds   r2, RI
0018      mov    r0, r2
001C      RETF
```

- POD always starts in COG mode, at the program entry point (which is COG address 1). When you first switch to LMM mode, POD will display the Catalina program entry point (**C_main**). In both COG and LMM modes you can single step **into** (F5) or single step **over** (F6) individual instructions, set breakpoints (SPACE or F9) and continue execution (F8). But note that single stepping **into**

a special LMM instruction does **not** mean to enter COG mode – it means enter an LMM subroutine if the LMM instruction is a CALL or CALI.

- Note that in LMM mode, it is possible to align the start of the assembly language display with one of the LMM instruction parameters, and not with the actual LMM instruction itself (e.g. the `long $00001000` shown in the example above, rather than the proceeding `LODL`). This can cause the assembly language to be decoded incorrectly.
- Whenever the program execution is paused, you can switch between LMM and COG modes. However, the results of single stepping in both modes during the same debug session are sometimes unpredictable (a bug I have yet to find!). Switching between LMM and COG mode to view memory or registers is fine, but it is better to avoid single stepping in both modes in the same debug session. Where possible single step in only one of the modes – i.e. only in LMM or only in COG mode, but not both.

SD Card Support

If you have a Propeller platform that has an SD Card (such as the Hybrid), you can use the SD Card in two ways:

- As a way of loading programs into the Propeller. Catalina provides a Generic SD Card Loader that can be used for this purpose.
- As a file system for Catalina programs to use¹⁷. Catalina provides targets specifically for this purpose. A description of the file system functions are given below in **File System Support**.

Note that the two uses are completely independent – a program may be loaded from an SD Card but not thereafter access the SD Card at all, or a program may be loaded from EEPROM or via serial I/O but then access the SD Card as a file system. Of course, a single program may also do both.

Also, note that if you remove and re-insert the SD Card at any time, you will need to restart any program that is using it.

More details and examples on using the SD Card are provided in the document **Getting Started with Catalina**.

Catalina also provides proxy SD drivers, which on multi-CPU systems allows one CPU to use an SD device physically connected to another CPU. This is further described in the **Multi-CPU System Support** section later in this document.

¹⁷ Note that some platforms (such as the Hydra) cannot use the SD Card at the same time as other hardware such as the SRAM card – this limits the usefulness of the SD Card as it can only be used to load and execute LMM programs, not XMM programs (which require the Hydra Xtreme SRAM card) - and LMM programs typically do not have enough space to load the SD Card file system support.

Real-Time Clock Support

On both the Propeller 1 and the Propeller 2, Catalina provides a plugin that implements a software real-time clock. This uses the Propeller's internal clock to count the elapsed time since the C program was started.

The standard C time functions described in the include file *time.h* (e.g. **clock()**, **time()** etc) all work as expected, and there are additional real-time clock functions described in *rtc.h*, including a function to set the time from a C program (see **rtc_settime()**).

When using the software real-time clock, if the time is not specifically set by the program, the clock will always start at **0**, and the time will always start at **1/1/1970 00:00:00**. The current time is only known by the plugin, and will be lost (i.e. reset to the initial value again) when the Propeller reboots.

The resolution of the **clock()** function is 1ms. For finer resolution, use the various propeller functions that access the Propeller counter directly (e.g. **_cnt()** or **_waitcnt()**).

The software real-time clock is enabled by specifying the Catalina symbol **CLOCK** (e.g. on the command line). For example, to compile the demo program */demos/examples/ex_time.c* to use the software real-time clock, use a command like:

```
catalina -lci ex_time.c -C CLOCK
```

File System Support

If you have a Propeller platform that has an SD Card (such as the Hybrid), Catalina provides full support for accessing FAT16 or FAT32 file systems on the SD Card¹⁸.

To enable full file system support, simply compile a program with one of the “extended” versions of the standard C library – i.e. **libcx** or **libcix**. The default version of the C library (**libc**) only supports I/O on **stdin**, **stdout** and **stderr**, whereas the extended versions allows I/O on files as well.

Catalina provides two sets of functions that can be used to access the file system:

1. The standard C library I/O functions described in the include file *stdio.h* (i.e. **fopen**, **fprintf**, **fscanf** etc).

Refer to any ANSI C language reference for details on the stdio functions – Catalina provides a full implementation of all functions documented in the ANSI C standard, except for the following note:

NOTE: *DOSFS does not allow a seek to a position beyond the current end of file. It is not well specified in the ANSI C standard whether the stdio function **fseek** must allow this or not, but it is certainly the case that Unix file systems support it, and the file will be padded with nulls if a write occurs at this*

¹⁸ FAT12 file systems can also be supported, but this is disabled in the libraries provided to save space (since FAT12 is rarely used any more). Support for FAT12 can be re-enabled, but this requires the Catalina library to be recompiled from source.

position. Programs that depend on this behavior may not perform correctly when compiled by Catalina. If in doubt, manually pad the file with nulls if a write has to be performed at a position beyond the current end of file.

2. The Catalina file system functions described in the include file *catalina_fs.h*. These functions are designed to be more space efficient than the standard C functions, and make it possible to write programs that can access the file system on a Propeller that has only the standard 32k of Hub RAM. Using the stdio functions generally requires XMM RAM.

The Catalina file system functions provide both *managed* and *unmanaged* functions for file I/O. The difference is how memory required for file control blocks is managed. The “managed” calls are simpler to use, because they allocate and manage internally the memory required for file control blocks – the downside is that these programs also pull in the **malloc** functions from the C library. This can incur a significant code size overhead on Propellers where the only RAM available is the internal 32kb of Hub RAM. For programs that do not want to incur this overhead, equivalent “unmanaged” functions are provided, which allow the use of statically allocated memory for file control blocks (note that if there is no “unmanaged” equivalent for a particular function, the function can be for both managed and unmanaged files).

The functions provided are as follows:

int _mount(int unit, int pnum)

mount must be called (once) before any file system access. (unit and pnum are normally left as zero). Note that only SD cards WITH AN MBR are supported.

int _unmount()

unmount must be called (once) before another SD card can be mounted.

int _create(const char *path, int mode)

create and open a new managed file (managed files have the FILEINFO structure allocated and managed internally). The file must be closed using the managed close function (i.e. **_close**). The mode can be:

0 - read only

1 - write only

2 - read and write

int _open(const char *path, int flags)

open a managed file and return the file number (managed files have the FILEINFO structure allocated and managed internally – which requires that **malloc** be used by the program). The file must be closed using the managed close function (i.e. **_close**). The mode can be:

0 - read only

1 - write only

2 - read and write

int _close(int d)

close a managed file (managed files have the FILEINFO structure allocated and managed internally). The file must have been opened using the managed open function (i.e. **_open**).

int _read(int d, char *buf, int nbytes)

read from a file.

int _write(int d, const char *buf, int nbytes)

write to a file.

off_t _lseek(int d, off_t offset, int whence)

seek (move) within a file. Whence can be:

0 – SEEK_SET (absolute position within the file)

1 – SEEK_CUR (relative to the current position within the file)

2 – SEEK_END (relative to the end of the file)

int _create_directory(const char *path)

create a new directory. The path to the new directory must already exist (i.e. only the last element of the path name is created) and a directory must not already exist with that name.

int _rename(const char *path, const char *newname)

rename a file from path to newname. The path is the complete path the the original file, but the new name is only the file name component – i.e. it should not contain the path again.

int _unlink(const char *path)

unlink (delete) a file.

int _create_unmanaged(const char *path, int mode, PFILEINFO fd)

create and open a new unmanaged file and return the file number (unmanaged files require a pointer to a FILEINFO structure to be provided). The file must be closed using the unmanaged close function (i.e. **_close_unmanaged**). The mode can be:

0 - read only

1 - write only

2 - read and write

int _open_unmanaged(const char *path, int flags, PFILEINFO fd)

open an unmanaged file and return the file number (unmanaged files require a pointer to a FILEINFO structure to be provided). The file must be closed using the unmanaged close function (i.e. **_close_unmanaged**). The mode can be:

- 0 - read only
- 1 - write only
- 2 - read and write

int _close_unmanaged(int d)

close an unmanaged file using the file number. Files that were opened unmanaged must be closed using the unmanaged close function (i.e. **_close_unamanged**).

Note that it is possible to mix the various file access functions – in fact sometimes it is necessary to do so. For example, the *stdio* functions provide no means of unmounting a file system if it becomes necessary to change SD cards – so the **_unmount** function provided can be used for this purpose. Similarly, the *stdio* functions provide no way of creating a new directory – so the Catalina file system library provides a **_create_directory** function that can be used.

Some important things to note about Catalina file system support:

Catalina only supports DOS 8.3 style file names (i.e. not long file names). Also, when specifying file names in the file functions, the 8 character name, the “.”, and the extension should all be specified, but some of the functions make accessible the raw FAT directory entries where 11 characters are always stored in each entry, with each of the 8 character name and 3 character extension components padded with trailing blanks and with no “.” inserted between them and no terminating character. To display a more “friendly” file name, it would be necessary to remove any trailing blanks and insert the “.” character.

Only FAT file systems with sector sizes of 512 bytes are supported. Some FAT file systems use larger sector sizes to increase the supported disk capacity – such file systems (which can be created under some versions of DOS and Windows) will be unusable under Catalina.

The path separator is “/” (even though FAT it is fundamentally a DOS file system where it might be more common to use “\” as a path separator) – so the following are valid **directory** names:

```
/
/dir
/dir1/dir2
```

The following are valid **file** names:

```
/my_file
```

```
/dir/xxx.txt
```

```
/dir1/dir2/xxx.bas
```

When creating files or directories, the path must already exist up to the final element – only the final element of the file or path name can actually be created in each call. Of course, repeated calls can be used to create deeper path names – e.g to create a file with path “/a/b/c/xxx.txt” in a blank file system:

first create directory “/a”

then create directory “/a/b”

then create directory “/a/b/c”

then create file “/a/b/c/xxx.txt”

There is no concept of “current directory”. If you want to refer to a file in a subdirectory, you must always specify the path all the way from the root directory (e.g. “/dir1/dir2/dir3/xxx.xxx”). There is a limit of 64 characters in any path name – this can be increased by recompiling the Catalina library from source, but this is not recommended as it increases the stack space required for all file access functions..

The line terminator is the UNIX line feed character, or “\n”. To create a file with DOS style line termination, it would be necessary to explicitly write a carriage return (“\r”) before each line terminator.

No checking is done in file names for characters that may be invalid in FAT file systems. This means it is possible to create files that would not be valid if the SD Card were to subsequently be used in a DOS or Windows system. For instance, “\” is mistakenly used as the path separator to try and create a file “xxx” in directory “dir1” by calling the file creation function with the path “dir1\xxx”, then the file system will instead create a file with the name “dir1\xxx” in the root directory - and this file may not be valid under MS-DOS or Windows since it will include the character “\” (which is invalid in FAT file systems).

When renaming files, no check is made to see if a file with the new name already exists – so it is possible to end up with two files of the same name in a directory. If this is possible, check that the new name does not already exist before renaming.

If the clock plugin is in use, using the stdio file functions will set the file **create** and **modify** time and date appropriately. If the year is 2000 or greater Catalina will assume this is correct (either because the hardware clock is in use, or the software clock time has been specifically set by the program) and use it. If not, the default time of **1/1/2006 01:01:00** will be used. Note that only the file **modify** time and date will be changed after file creation - the **access** date is never changed after file creation, because it would slow down the file system too much to do so on every file access.

Serial Device Support

Catalina has many options for support for serial devices:

1. Use the **PC** HMI plugin, which configures a single serial port, normally on pins 30 & 31 and at 115200 baud. You use this option by adding **-C PC** on the command line or in Geany. Access to this single serial port is via the standard C stdio functions (e.g. **getc**, **putc**, **scanf**, **printf** etc). To configure this port, you can edit the details in the appropriate **<platform>_DEF.inc** file for your platform (e.g. **HYDRA_DEF.inc**).

Because it is specifically designed to implement various HMI specific capabilities, using the PC HMI plugin as a serial option requires 3 cogs for 1 port, so it is mainly worthwhile if you want to access the serial port via stdio (or you have a program that does so and do not want to change it).

2. Use the **TTY** HMI plugin, which configures a single serial port, normally on pins 30 & 31 and at 115200 baud. This option uses a 16 byte transmit and receive buffer. You use this option by adding **-C TTY** on the command line or in Geany. Access to this single serial port is via the standard C stdio functions (e.g. **getc**, **putc**, **scanf**, **printf** etc). To configure this port, you can edit the details in the appropriate **<platform>_DEF.inc** file for your platform (e.g. **HYDRA_DEF.inc**).

The **TTY** HMI option uses one less cog than the PC HMI option, but still requires two cogs – it is worthwhile if you want to access the serial port via stdio (or you have a program that does so and do not want to change it).

3. Use the **TTY256** HMI plugin, which configures a single serial port, normally on pins 30 & 31 and at 115200 baud. This option uses a 256 byte transmit and receive buffer. You use this option by adding **-C TTY256** on the command line or in Geany. Access to this single serial port is via the standard C stdio functions (e.g. **getc**, **putc**, **scanf**, **printf** etc). To configure this port, you can edit the details in the appropriate **<platform>_DEF.inc** file for your platform (e.g. **HYDRA_DEF.inc**).

The **TTY256** HMI option uses one less cog than the PC HMI option, but still requires two cogs – it is worthwhile if you want to access the serial port via stdio (or you have a program that does so and do not want to change it).

4. Use the **4 port Serial** library, which allows up to 4 serial ports using one cog, on any pins. You use this option by adding **-lserial4** on the command line or in Geany. This loads the 4 port serial plugin, and links your program with the **libserial4** library. Access to these serial ports are via special library functions, described below. To configure these ports, you can edit the details in **Extras.spin**. This is the recommended way if you only have one spare cog, or if you need multiple serial ports.

Note that by default, the file **Extras.spin** file configures only one serial port (port 0) on pins 30 & 31 at 115200 baud. These are the same pins as the **PC** HMI normally uses, so if your propeller platform enables the PC HMI plugin by

default, you will need to add the **-C NO_HMI** flag to use the serial port (or edit **Extras.spin** to move the port to other pins).

5. Use the **tty** serial library, which allows access to a single high speed serial port using one cog, on any pins. You use this option by adding **-ltty** on the command line, or adding this option in Geany. This loads the **tty** plugin, and links your program with the **libtty** library. Access to the serial ports are via special library functions, described below. To configure this port, you can edit the details in **Extras.spin**. This is the recommended way if you only have one spare cog, and need access to a high speed serial port.

Note that by default, the file **Extras.spin** file configures the **tty** serial port on pins 30 & 31 at 115200 baud. These are the same pins as the **PC** HMI normally uses, so if your propeller platform enables the PC HMI plugin by default, you will need to add the **-C NO_HMI** flag to use the **tty** serial port (or edit **Extras.spin** to move the port to other pins).

6. Use the **tty256** serial library, which allows access to a single high speed serial port using one cog, on any pins. The **tty256** library is similar to the **tty** library, except the serial plugin used supports 256 byte buffers for Tx and Rx, which may be required for some serial applications. You use this option by adding **-ltty256** on the command line, or adding this library in Geany. This loads the **tty256** plugin, and links your program with the **libtty256** library. Access to the serial ports are via special library functions, described below. To configure this port, you can edit the details in **Extras.spin**. This is the recommended way if you only have one spare cog, and need access to a high speed serial port that supports Tx and Rx buffers of 256 bytes.

Note that by default, the file **Extras.spin** file configures the **tty256** serial port on pins 30 & 31 at 115200 baud. These are the same pins as the **PC** HMI normally uses, so if your propeller platform enables the PC HMI plugin by default, you will need to add the **-C NO_HMI** flag to use the **tty256** serial port (or edit **Extras.spin** to move the port to other pins).

You can combine several methods of adding serial capabilities, provided you have sufficient cogs and the pin configurations used by the various serial ports don't conflict. Or you can use another HMI option (e.g. TV, VGA) and the serial plugins together.

The **tty** library (**libtty**)

Here are the functions implemented in **libtty** (also described in the include file *catalina_tty.h*). They are designed to be equivalent to the Spin functions defined in the original Full Duplex Serial driver:

int tty_rxflush()

Flush the receive buffer until empty (discard any characters received).

int tty_rxcheck()

Check if there are characters in the receive buffer – returns the byte, or -1 if no characters are available. Does not wait.

int tty_rxtime(unsigned ms)

Wait up to **ms** milliseconds, or until a character is received. Returns -1 if no character was received in the specified time.

int tty_rx()

Read a byte from the receive buffer. If there are no characters, wait until a character is received.

int tty_tx(char txbyte)

Put a byte in the transmit buffer. If there is no space, in the buffer wait until there is space.

int tty_txflush()

Wait until there are no characters in the transmit buffer (i.e. all bytes have been sent).

int tty_txcheck(unsigned port)

Return the number of character spaces available in the transmit buffer – a result of zero (or less) means a call to **tty_tx()** would block.

void tty_str(char *stringptr)

Send a null-terminated string to the transmit buffer.

void tty_decl(int value, int digits, int flag)

Send a signed decimal number. This function is not usually called directly – instead, call **dec**, **decf** or **decx** (see below). The **flag** has the following meaning:

- 0** print only required characters (plus sign if necessary). Note that **digits** should always be specified as 10 in this case.
- 1** right justify and space pad up number using **digits** characters (plus sign if necessary).
- 2** right justify and zero pad number using **digits** characters (plus sign if necessary).

void tty_hex(unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal.

void tty_ihex(unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal, prefixed by a '\$' character.

void tty_bin(unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary.

void tty_ibin(unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary, prefixed by a '%' character.

void tty_padchar(unsigned count, char txbyte)

Send **count** instances of the character **txbyte**.

The following useful macros are defined to assist in the use of the above functions:

tty_dec(value)

Send a signed decimal string, up to 10 digits, plus a sign character if required.

tty_decf(value, width)

Send a space-padded fixed width decimal string, in **width** digits, plus a sign character if required. The **width** can be up to 10.

tty_decx(value, width)

Send a zero-padded fixed width decimal string, in **width** digits, plus a sign character if required. The **width** can be up to 10.

tty_putc(txbyte)

Same as **tty_tx**

tty_newline(port)

Send a newline character. By default, the newline is ASCII 10. If the C symbol **TTY_CR_NEWLINE** is defined, then the newline is ASCII 13. The purpose of defining this symbol is to make this function work like the Spin equivalent.

tty_strln(port, stringptr)

Send a zero terminated string newline character. By default, the newline is ASCII 10. If the C symbol **TTY_CR_NEWLINE** is defined, then the newline is ASCII 13. The purpose of defining this symbol is to make this function work like the Spin equivalent.

tty_cls(port)

Send a Form Feed character (ASCII 12).

tty_getc(port)

Same as **tty_rx**.

The tty256 library (libtty256)

The functions implemented in **libtty256** are identical to those described above in **libtty** (also described in the include file *catalina_tty.h*).

The 4 port Serial library (libserial4)

Here are the functions implemented in **libserial4** (also described in the include file *catalina_serial4.h*). They are designed to be equivalent to the Spin functions defined in the original Spin version of the 4 port serial driver, and also to the **libtty** functions described above. In all cases, the **port** number is a number in the range 0 .. 3:

int s4_rxflush(unsigned port)

Flush the receive buffer until empty (discard any characters received).

int s4_rxcheck(unsigned port)

Check if there are characters in the receive buffer – returns the byte, or -1 if no characters are available. Does not wait.

int s4_rxtime(unsigned port, unsigned ms)

Wait up to **ms** milliseconds, or until a character is received. Returns -1 if no character was received in the specified time.

int s4_rxcount(unsigned port)

Returns the number of characters currently waiting in the receive buffer.

int s4_rx(unsigned port)

Read a byte from the receive buffer. If there are no character, wait until a character is received.

int s4_tx(unsigned port, char txbyte)

Put a byte in the transmit buffer. If there is no space, in the buffer wait until there is space.

int s4_txflush(unsigned port)

Wait until there are no characters in the transmit buffer (i.e. all bytes have been sent).

int s4_txcheck(unsigned port)

Return the number of character spaces available in the transmit buffer – a result of zero (or less) means a call to **s4_tx()** would block.

int s4_txcount(unsigned port)

Returns the number of characters currently waiting in the transmit buffer.

void s4_str(unsigned port, char *stringptr)

Send a null-terminated string to the transmit buffer.

void s4_decl(unsigned port, int value, int digits, int flag)

Send a signed decimal number. This function is not usually called directly – instead, call **dec**, **decf** or **decx** (see below). The **flag** has the following meaning:

- 0** print only required characters (plus sign if necessary). Note that **digits** should always be specified as 10 in this case.
- 1** right justify and space pad up number using **digits** characters (plus sign if necessary).
- 2** right justify and zero pad number using **digits** characters (plus sign if necessary).

void s4_hex(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal.

void s4_ihex(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal, prefixed by a '\$' character.

void s4_bin(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary.

void s4_ibin(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary, prefixed by a '%' character.

void s4_padchar(unsigned port, unsigned count, char txbyte)

Send **count** instances of the character **txbyte**.

The following useful macros are defined to assist in the use of the above functions:

s4_dec(port, value)

Send a signed decimal string, up to 10 digits, plus a sign character if required.

s4_decf(port, value, width)

Send a space-padded fixed width decimal string, in **width** digits, plus a sign character if required. The **width** can be up to 10.

s4_decx(port, value, width)

Send a zero-padded fixed width decimal string, in **width** digits, plus a sign character if required. The **width** can be up to 10.

s4_putc(port, txbyte)

Same as **s4_tx**

s4_newline(port)

Send a newline character. By default, the newline is ASCII 10. If the C symbol **S4_CR_NEWLINE** is defined, then the newline is ASCII 13.

The purpose of defining this symbol is to make this function work like the Spin equivalent.

s4_strln(port, stringptr)

Send a zero terminated string newline character. By default, the newline is ASCII 10. If the C symbol **S4_CR_NEWLINE** is defined, then the newline is ASCII 13. The purpose of defining this symbol is to make this function work like the Spin equivalent.

s4_cls(port)

Send a Form Feed character (ASCII 12).

s4_getc(port)

Same as **s4_rx**.

Sound Support

Sound support is provided by a plugin based on Nick Sabalausky's 22KHz, 16-bit, 6 Channels Sound Driver. This sound driver works on many platforms, including the **Hydra** and **C3**. It is configured by setting the **SOUND_PIN** in the platform definition file (e.g. *C3_DEF.inc*)

Here are the functions implemented in the **libsound** library (also described in the include file *catalina_sound.h*):

```
extern void PlaySoundFM (int channel,  
                            unsigned int shape,  
                            unsigned int freq,  
                            unsigned int duration,  
                            unsigned int volume,  
                            unsigned int amp_env)
```

Starts playing a frequency modulation sound. If a sound is already playing, then the old sound stops and the new sound is played.

channel: The channel on which to play the sound (0-5)

shape: The desired shape of the sound. Use any of the following constants: **SHAPE_SINE**, **SHAPE_SAWTOOTH**, **SHAPE_SQUARE**, **SHAPE_TRIANGLE**, **SHAPE_NOISE** (see *catalina_sound.h*). Do NOT send a **SHAPE_PCM_*** constant, use **PlaySoundPCM()** instead.

freq: The desired sound frequency. Can be a number or a **NOTE_*** constant (see *catalina_sound.h*). A value of 0 leaves the frequency unchanged.

duration: Either a 31-bit duration to play sound for a specific length of time, or (**DURATION_INFINITE** | "31-bit duration of amplitude envelope") to play until **StopSound**, **ReleaseSound** or another call to **PlaySound** is called.

See "Explanation of Envelopes and Duration" in *catalina_sound.h* for important details.

volume: The desired volume (1-255). A value of 0 leaves the volume unchanged.

amp_env: The amplitude envelope, specified as eight 4-bit nybbles from \$0 (0% of arg_volume, no sound) to \$F (100% of arg_volume, full volume), to be applied least significant nybble first and most significant nybble last. Or, use **NO_ENVELOPE** to not use an envelope. See "Explanation of Envelopes and Duration" in *catalina_sound.h* for important details.

**extern void PlaySoundPCM(int channel,
void *pcm_start,
void *pcm_end,
unsigned int volume)**

Plays a signed 8-bit 11KHz PCM sound once. If a sound is already playing, then the old sound stops and the new sound is played.

channel: The channel on which to play the sound (0-8)

pcm_start: The address of the PCM buffer

pcm_end: The address of the end of the PCM buffer

volume: The desired volume (1-255)

amp_env: The amplitude envelope, specified as eight 4-bit nybbles from \$0 (0% of arg_volume, no sound) to \$F (100% of arg_volume, full volume), to be applied least significant nybble first and most significant nybble last. Or, use **NO_ENVELOPE** to not use an envelope. See "Explanation of Envelopes and Duration" in *catalina_sound.h* for Important details.

extern void StopSound(int channel)

Stops playing a sound.

channel: The channel to stop.

extern void ReleaseSound(int channel)

"Releases" an infinite duration sound - .i.e. starts the release portion of the sound's amplitude envelope.

channel: The channel to "release".

extern void SetFreq(int channel, unsigned int freq)

Changes the frequency of the playing sound. If called repeatedly, it can be used to create a frequency sweep.

channel: The channel to set the frequency of.

freq: The desired sound frequency. Can be a number or a **NOTE_*** constant (see *catalina_sound.h*). A value of 0 leaves the frequency unchanged.

extern void SetVolume(int channel, unsigned int volume)

Changes the volume of the playing sound. If called repeatedly, it can be used to manually create an envelope.

channel: The channel to set the volume of.

volume: The desired volume (1-255). A value of 0 leaves the volume unchanged.

SPI/I2C Support

Simple SPI & I2C bus support is provided by a plugin based on Mike Green's **sdspiFemto** Spin object. It supports both an I2C and SPI Bus. The I2C bus is intended for use in communicating with an EEPROM, and the SPI bus is intended for use communicating with an SD Card.

The EEPROM functions should work on all Propellers, and the SD Card functions should work with the SD Cards on all Propellers except the C3 (which requires additional select logic to select amongst various devices that share the SPI bus - however, the existing Catalina SD Plugin supports the C3).

The library provides functions to read and write EEPROMs or SD Cards, and to boot a program from an address in EEPROM, or from a sector on SD Card.

The following library routines are provided in **libspi**. These library routines allocate a lock to prevent contention, so they are safe for use with a multithreading kernel:

extern int spi_bootEEPROM(unsigned int addr)

Load and run a program from EEPROM.

addr the address in EEPROM (use the **EEPROM_ADDR** macro to encode the address – see *catalina_spi.h*)

NOTE: This function will return any lock allocated by the SPI code to the pool of unused locks, but it is the caller's responsibility to ensure that any OTHER locks checked out are returned before calling the boot function – otherwise if the program being booted uses locks, it might not be able to allocate one!

extern int spi_readEEPROM(unsigned int addr, void *buffer, int count)

Read from EEPROM To buffer

addr address in EEPROM (use the **EEPROM_ADDR** macro to encode the address – see *catalina_spi.h*)

buffer buffer to read

count count of bytes to read

extern int spi_writeEEPROM(unsigned int addr, void *buffer, int count)

Write from buffer to EEPROM

addr address in EEPROM (use the **EEPROM_ADDR** macro to encode the address – see *catalina_spi.h*)

buffer buffer to write

count count of bytes to write

extern int spi_checkPresence(unsigned int addr)

Check there is an I2C bus and EEPROM at the specified address. Note that this routine cannot distinguish between a 32Kx8 and a 64Kx8 EEPROM since the 16th address bit is a "don't care" for the 32Kx8 devices.

Return true if EEPROM present, false otherwise.

addr address in EEPROM to check (use the **EEPROM_ADDR** macro to encode the address – see *catalina_spi.h*)

extern int spi_writeWait(unsigned int addr)

Wait for EEPROM Write to finish.

addr address to check

Return true if EEPROM present, false otherwise.

extern int spi_initSDCard(int DO, int Clk, int DI, int CS)

Initialize SD Card

DO, Clk, DI, CS Pin numbers to use

extern int spi_stopSDCard(void)

Stop SD Card access

extern int spi_bootSDCard(unsigned int addr, int count)

Boot from an SD Card

addr the address on the SDCard.

count the count of bytes to load (must be at least 16)

NOTE: This function will return any lock allocated by the SPI code to the pool of unused locks, but it is the caller's responsibility to ensure that any OTHER locks checked out are returned before calling the boot function – otherwise if the program being booted uses locks, it might not be able to allocate one!

extern int spi_readSDCard(unsigned int addr, void *buffer, int count)

Read from SDCard to buffer

addr address on SDCard
 buffer buffer to read
 count count of bytes to read

extern int spi_writeSDCard(unsigned int addr, void *buffer, int count)

write from buffer to SDCard

addr : address on SDCard
 buffer : buffer to write
 count : count of bytes to write

extern unsigned int spi_getControl(int i)

Get an unsigned int from the control block

i control block index

extern void spi_setControl(int i, unsigned int value)

Set a value in the control block

i control block index
 value value to set

The following macros are provided in *catalina_spi.h* to assist in constructing EEPROM addresses:

EEPROM_ADDR(SCL_PIN, ADDR)

Format an EEPROM address for use in the EEPROM functions such as **spi_readEEPROM()** or **spi_writeEEPROM()**

SCL_PIN the number of the I2C bus SCL pin (the SDA pin is assumed to be SCL_PIN + 1)

ADDR The address (up to 19 bits) within the EEPROM.

SPI Flash and Cache Support

This version of Catalina supports loading and executing XMM programs from SPI RAM and SPI Flash.

When using serial RAM such as SPI RAM or SPI Flash (which is the only XMM RAM available on the **C3**, **SuperQuad**, **RamPage**, **RamPage2** and **Propeller Memory Card** boards) executing programs can be very slow. To speed this up, Catalina now provides a caching XMM driver. This driver dedicates a portion of Hub RAM to “cache” the contents of the XMM RAM, so that the XMM RAM need only be consulted if the data is not already available in the much faster Hub RAM.

Although the caching driver can be used in conjunction with *any* supported XMM platform, it is mainly intended for platforms that use SPI RAM or SPI Flash. On those platforms, it can speed up program execution by a factor of 10. While it can also

speed up program execution on other platforms with slow XMM RAM (such as the DracBlade), the performance improvement is less dramatic. On some XMM platforms (such as the Hydra or Hybrid) it results in only a small improvement, and on platforms with very fast XMM (such as the RamBlade) using the caching driver can actually *slow down* performance.

On platforms with both SPI RAM and SPI Flash, the caching driver does not **need** to be used to use only the SPI RAM – but it **must** be used in order to use the SPI Flash.

The caching XMM driver can be enabled by defining one of the following symbols on the command line:

<code>CACHED_1K</code>	use a 1k cache
<code>CACHED_2K</code>	use a 2k cache
<code>CACHED_4K</code>	use a 4k cache
<code>CACHED_8K</code>	use a 8k cache
<code>CACHED</code>	use a default size (8K) cache

For example:

```
catalina hello_world.c -lc -C CACHED
```

Note that the caching XMM driver requires sufficient free hub memory according to the size of the cache, and also an extra cog.

Since SPI Flash is non-volatile, it is possible to execute programs loaded into SPI Flash without having to reload them. To facilitate this, Catalina provides a **Flash_Boot** utility, specifically designed to boot programs already loaded into SPI Flash. Note that when compiling the **Flash_Boot** utility, you need to specify ALL the plugins you want – only plugins specified will be loaded. For example, if you want the **Flash_Boot** program to load an **SD** driver, a **CLOCK** driver and use the **HIRES_vGA** HMI but not load a mouse, you would need to compile the utilities and specify the following options during the build when prompted:

```
SD CLOCK HIRES_VGA NO_MOUSE
```

To run a program loaded into SPI Flash on Propeller reset, you can program the **Flash_Boot** binary into EEPROM.

You can also recompile the **Flash_Boot** program with different drivers – this might (for example) enable you to run the program in SPI Flash with either a TV driver or a VGA driver.

Note that the program in Flash will NOT be aware of whether or not the caching driver is in use, or of the size of the cache. This means that the same program can be easily executed with different cache sizes using different versions of **Flash_Boot**.

Lua Support

Catalina offers extensive support for the Lua scripting language.

What is **Lua**? The following description is from the **Lua Reference Manual** (<https://www.lua.org/manual/5.4/manual.html>):

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode with a register-based virtual machine, and has automatic memory management with a generational garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

*Lua is implemented as a library, written in clean C, the common subset of standard C and C++. The Lua distribution includes a host program called **lua**, which uses the Lua library to offer a complete, standalone Lua interpreter, for interactive or batch use. Lua is intended to be used both as a powerful, lightweight, embeddable scripting language for any program that needs one, and as a powerful but lightweight and efficient stand-alone language.*

*As an extension language, Lua has no notion of a "main" program: it works embedded in a host client, called the embedding program or simply the host. (Frequently, this host is the stand-alone **lua** program.) The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework.*

Catalina supports Lua 5.4 on both the Propeller 1 and 2. On the Propeller 1, Lua is supported only in the XMM LARGE memory model, and the size of Lua programs is limited - but on the Propeller 2 it is supported in *all* memory models, and executing large Lua programs is perfectly feasible.

Lua is supported in multiple ways by both Catalina compiler and the Catalyst operating system:

1. Lua scripts can be executed directly from the Catalyst Command line. In fact, some Catalyst commands are implemented as Lua scripts. **See the Catalyst Reference Manual.**
2. The payload program loader adds the capability to use Lua scripts to interact with the Propeller. Catalina uses this for its own compiler validation - see the **README.TXT** in the *validation* folder.

3. Catalina adds various low-level Propeller-specific functions and also multi-processing capabilities to Lua - see the document **Lua on the Propeller 2 with Catalina** (the multi-processing **threads** module described in this document is available only on the Propeller 2, but the low-level functions in the **propeller** module are available on both the Propeller 1 and 2).
4. Lua is built into the Catalina libraries, which means that Lua scripts can easily be embedded in Catalina C programs. See the section below for an example.

Embedding Lua scripting in a C program

The following is a complete working example of embedding a Lua script in a Catalina C program. A slightly more elaborate version can be found as *lscript.c* in the folder *demos/lua*:

```
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

int main(int argc, char *argv[]) {
    int result;

    // create a new Lua state and open the standard libraries
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // execute the script contained in the file "script.lua"
    result = luaL_dofile(L, "script.lua");
}
```

To compile this program, use a command like:

```
catalina -C LARGE -C CACHED_1K -lcx -lm -llua lscript.c linit.c -W-w
```

Depending on the platform, you may need to add `-c FLASH` and other options. For example, on the C3 you might use a command like:

```
catalina -C LARGE -C CACHED_1K -C FLASH -C TTY -lcx -lm -llua
lscript.c linit.c -W-w
```

Note that this command includes `-llua`, which tells Catalina to use the Lua library, and also compiles the file *linit.c*, which loads the Lua libraries. This is the only Lua file not contained in the Lua library, because it is often desirable to customize the Lua libraries that are loaded by such a program. The file *linit.c* is only required during compilation.

You can modify the *script.lua* file to contain any valid Lua program. Here is what is in the *script.lua* provided in *demos/lua*:

```
# a simple Lua script

print('Hello, World (from "script.lua")\n');

io.write("enter number A: ");
```

```
A = io.read("n");  
  
io.write("enter number B: ");  
B = io.read("n");  
  
io.write("A + B = " .. A + B .. "\n");
```

To execute the program, load both *lscript.bin* and *script.lua* onto an SD card containing Catalyst. Then execute *lscript.bin* from the Catalyst command line:

lscript

Finally, note that the precompiled version of Lua that comes with Catalyst can also be used to execute the Lua script. From the Catalyst command line, execute:

lua script.lua

or just

script

Catalina Targets

A Catalina *target* is a SPIN program responsible for establishing the execution environment for Catalina programs. The precise details of the target are often unknown to the Catalina program, and may depend on the underlying Propeller platform. For example, the same C program can be executed using different targets – one target may use a TV as its display device, and another – even on the same platform – may use the VGA display for the same purpose. Wrapping such details up in a target gives Catalina programs an effective hardware abstraction layer.

Catalina Propeller 1 Targets

This section describes the targets provided in the standard Catalina Target Package for the Propeller 1. Eleven such targets are provided – three for LMM programs, and two each for CMM, EMM, SMM and XMM programs:

- The default LMM target (*lmm_default*). Any program compiled with the **-x0** command line option (or without any **-x** option) will use this target unless another target is explicitly specified using the **-t** command line option.
- The POD debugger LMM target (*lmm_debug*). This target supports the POD debugger. This target must be explicitly specified on the command line using the **-tdebug** option.
- The BlackBox debugger LMM target (*lmm_blackcat*). This target supports the BlackBox debugger. This target is used whenever the **-g** or **-g3** command line options are specified.
- The default CMM target (*cmm_default*). Any program compiled with the **-C COMPACT** command line symbol (or **-x8**) will use this target unless another target is explicitly specified using the **-t** command line option.
- The BlackBox CMM target (*cmm_blackcat*). This target supports the BlackBox debugger. This target is used whenever the **-g** or **-g3** command line options are specified (in conjunction with **-C COMPACT** or the **-x8** option).
- The default EMM target (*emm_default*). Any TINY LMM program compiled with the **-C EEPROM** command line symbol (or **-x1**) will use this target unless another target is explicitly specified using the **-t** command line option. More detail on EMM is given in the section **EMM support** below.
- The BlackBox EMM target (*emm_blackcat*). This target supports the BlackBox debugger. This target is used whenever the **-g** or **-g3** command line options are specified (in conjunction with **-C EEPROM** or the **-x1** option).
- The default SMM target (*smm_default*). Any program compiled with the **-C SDCARD** command line symbol (or **-x6**) will use this target unless another target is explicitly specified using the **-t** command line option. More detail on SMM is given in the section **SMM support** below.

- The BlackBox SMM target (*smm_blackcat*). This target supports the BlackBox debugger. This target is used whenever the **-g** or **-g3** command line options are specified (in conjunction with **-C EEPROM** or the **-x6** option).
- The default XMM target (*xmm_default*). Any program compiled with the **-C SMALL** or **-C LARGE** command line symbol (or the **-x2** or **-x5** options) will use this target unless another target is explicitly specified using the **-t** command line option. More detail on XMM is given in the section **XMM Support** below.
- The BlackBox XMM target (*xmm_blackcat*). This target supports the BlackBox debugger. This target is used whenever the **-g** or **-g3** command line options are specified (in conjunction with **-C SMALL** or **-C LARGE** (or the **-x2** or **-x5** options).

Each target is a SPIN program loaded from the **target** directory, and compiled along with the C program by Catalina.

The configuration options supported by the targets in the standard Catalina Target Package are given in the following section.

Default Target Configuration Options

For the Propeller 1, the standard Catalina Target Package supports the **HYDRA**, the **HYBRID**, the Parallax **ACTIVITY** board, the Parallax **DEMO** board, the Parallax **FLIP** module, the **RAMBLADE**, the **RAMBLADE3**, the **TRIBLADEPROP**, the **DRACBLADE**, **ASC**, **C3**, **PP** and **QUICKSTART** boards. It also supports XMM add-on board such as the **SUPERQUAD**, **RAMPAGE**, **RP2** and **PMC** boards.

The following symbols can be defined on the command line to affect the configuration of the default target package (note that some symbols only apply to specific targets or memory modes):

HYDRA	use HYDRA pin definitions, drivers and XMM functions
HYBRID	use HYBRID pin definitions, drivers and XMM functions
DEMO	use DEMO board pin definitions and drivers (this platform has no XMM support)
DRACBLADE	use DRACBLADE pin definitions, drivers and XMM functions
RAMBLADE	use RAMBLADE pin definitions, drivers and XMM functions
TRIBLADEPROP	use TRIBLADEPROP pin definitions, drivers and XMM functions
ACTIVITY	use ACTIVITY pin definitions and drivers (this platform has no XMM support)
FLIP	use FLIP pin definitions and drivers (this platform has no XMM support)

ASC	use ASC board pin definitions and drivers (this platform has no XMM support)
C3	use C3 pin definitions, drivers and XMM functions
PP	use PROPELLER PLATFORM pin definitions, drivers and XMM functions
QUICKSTART	use QUICKSTART pin definitions, drivers and XMM functions
CUSTOM	use a user-customized set of pin definitions, drivers and XMM functions (if applicable)
HX512	use HX512 pin definitions, drivers and XMM functions
SUPERQUAD	use SUPERQUAD pin definitions, drivers and XMM functions
RAMPAGE	use RAMPAGE pin definitions, drivers and XMM functions
RP2	use RAMPAGE 2 pin definitions, drivers and XMM functions
PMC	use PROPELLER MEMORY CARD pin definitions, drivers and XMM functions
CPU_1	on the TRIBLADEPROP, this means to use CPU #1 pin definitions and XMM functions – if not specified, CPU #2 XMM functions are used by default
CPU_2	on the TRIBLADEPROP, this means to use CPU #2 pin definitions and XMM functions (this is also the default)
CPU_3	on the TRIBLADEPROP, this means to use CPU #3 pin definitions and devices.
COMPACT	compile a CMM program (COMPACT memory model)
TINY	compile an LMM program to use the TINY memory model
SMALL	compile an XMM program to use the SMALL memory model
LARGE	compile an XMM program to use the LARGE memory model
ALTERNATE	use the alternate LMM Kernel (the alternate kernel is slightly smaller in size but does not include any floating point support).
SDCARD	use the SDCARD two-phase loader
EEPROM	use the EEPROM two-phase loader
FLASH	use an SPI FLASH loader
HIRES_VGA	load a High Resolution VGA driver (not supported on the HYBRID, RAMBLADE or RAMBLADE3)
LORES_VGA	load a Low Resolution VGA driver (not supported on the RAMBLADE or RAMBLADE3)

VGA	(same as LORES_VGA)
HIRES_TV	load a High Resolution TV driver (not supported on the RAMBLADE or RAMBLADE3)
LORES_TV	load a Low Resolution TV driver (not supported on the RAMBLADE or RAMBLADE3)
TV	(same as LORES_TV)
NTSC	use NTSC mode (TV drivers only)
NO_INTERLACE	use non-interlace mode (TV drivers only)
PC	load a PC terminal emulator HMI plugin with screen and keyboard support
PROPTERMINAL	load a PropTerminal HMI plugin with screen, keyboard and mouse support.
TTY	load a simple serial HMI plugin with screen and keyboard support (no proxy support).
TTY256	load a 256 byte buffer version of the serial HMI plugin with screen and keyboard support (no proxy support).
CR_ON_LF	Translate LF to CR LF on output
NO_CR_TO_LF	Disable translation of CR to LF on input
NON_ANSI_HMI	Disable ANSI compliance in HMI (revert to prior behavior)
CLOCK	load a Real-Time Clock plugin (or enable the RTC functionality in the SD plugin if it is loaded)
SD	load the SD plugin (this is not usually required, since it is implied by the -lcx and -lcix options)
NO_FP	do not load any Floating Point plugins (even if implied by other options)
NO_FLOAT	same as NO_FP
NO_HMI	do not load any HMI plugin (even if implied by other options)
NO_MOUSE	do not start a mouse driver (even if one is loaded)
NO_KEYBOARD	do not start a keyboard driver (even if one is loaded)
NO_SCREEN	do not start a screen driver (even if one is loaded)
PROXY_SD	See the Proxy Devices section later in this document
PROXY_SCREEN	See the Proxy Devices section later in this document
PROXY_MOUSE	See the Proxy Devices section later in this document
PROXY_KEYBOARD	See the Proxy Devices section later in this document

CACHED_1K	Use a 1K cache for XMM access
CACHED_2K	Use a 2K cache for XMM access
CACHED_4K	Use a 4K cache for XMM access
CACHED_8K	Use a 8K cache for XMM access
CACHED	Same as CACHED_8K
GAMEPAD	Include the Gamepad driver
DISABLE_REBOOT	Disable the automatic reboot if the program exits from the main() function – useful if you are using the TV or VGA driver and want the screen output to remain when the program exits.

Symbols are defined on the command line using the **-C** option. Multiple symbols can be defined, but **-C** must be specified before each one. For example, to compile a program for the Hybrid using a high-resolution TV driver in NTSC mode with clock support you might use a command like:

```
catalina test_time.c -lc -C HYBRID -C HIRES_TV -C NTSC -C CLOCK
```

Because using multiple symbols to specify the configuration of the target is so common, there is a better way to specify them if you intend using the same configuration for many compilations – set the **CATALINA_DEFINE** environment variable, as follows (under Windows):

```
set CATALINA_DEFINE=HYBRID HIRES_TV NTSC CLOCK
```

or as follows (under Linux if using the bash shell):

```
CATALINA_DEFINE="HYBRID HIRES_TV NTSC CLOCK"; export CATALINA_DEFINE
```

Then a command such as:

```
catalina test_time.c -lc
```

has the same effect as specifying all the symbols using **-C** on the command line (note that the **-C** option should not be specified for symbols defined using the environment variable).

Note that you cannot specify the same symbol both in an environment variable and on the command line – doing so will result in an error message to the effect that the symbol is already defined.

Knowing what the current setting of the various Catalina environment variables is can be very important. To display the current settings, use the command **catalina_env** – in the above case you might see output like:

```
CATALINA_DEFINE    = HYBRID HIRES_TV NTSC CLOCK
CATALINA_INCLUDE  = [default]
CATALINA_LIBRARY  = [default]
CATALINA_TARGET   = [default]
CATALINA_LCCOPT   = [default]
CATALINA_TEMPDIR  = [default]
LCCDIR            = [default]
```

To unset an environment variable, use a command such as:

```
unset CATALINA_TARGET
```

or (Windows only):

```
set CATALINA_TARGET=
```

Catalina Hub Resource Usage

On the Propeller 1, Catalina uses some of the upper Hub RAM for the registry, the debugger (if in use), the cache (if in use), the argument list, the program loader, and various items of configuration data. This is described in detail in *target\p1\Catalina_Common.inc*.

LMM Support

All Propeller platforms are capable of using the Propeller in LMM (**L**arge **M**emory **M**odel) mode to support C programs of up to 32 kb. Catalina provides an LMM Kernel for executing such programs.

The memory model used for LMM support is fairly simple, with all program code and data held in Hub RAM, and no real need to differentiate between code and data. Supporting XMM programs is more complex. To do this, Catalina divides each program into 4 program segments:

- Code** : a read-only segment containing program code
- Cnst** : a read-only segment containing constant data
- Init** : a read/write segment containing static data
- Data** : a read/write segment containing dynamic data

This type of segmentation is fairly standard to all compilers (although for historical reasons some use different segment names, such as **Text** instead of **Cnst**, or **BSS** instead of **Data**). The Catalina Binder separates out and groups together all the different items of code and data according to the segments they belong to.

By default (i.e. if no specific command line options specify the contrary), Catalina compiles programs as LMM programs with the Catalina LMM Kernel and all segments combined into 32k (this is the **TINY** memory model). These programs can run on *any* Propeller.

CMM Support

All Propeller platforms are capable of using the Propeller in CMM (**C**ompact **M**emory **M**odel) mode to support C programs of up to 32kb. Catalina provides a CMM Kernel for executing such programs.

The memory model used for CMM support is fairly simple, with all program code and data held in Hub RAM, and no real need to differentiate between code and data. To do this, Catalina divides each program into 4 program segments:

- Code** : a read-only segment containing program code
- Cnst** : a read-only segment containing constant data
- Init** : a read/write segment containing static data
- Data** : a read/write segment containing dynamic data

The Compact Memory Model is a hybrid kernel – it uses some LMM techniques, and some techniques that make it more like the Parallax Spin kernel – this is what allows it to generate code sizes that are often less than **half** the equivalent LMM code sizes. The tradeoff is that CMM programs are slower than LMM programs (although still faster than Spin).

XMM Support

On the Propeller 1, Catalina also provides XMM (eXternal Memory Model) support for executing C programs *larger* than 32kb – but this requires either a Propeller equipped with additional RAM hardware, or a Propeller with an EEPROM larger than 32kb (see the section **XEPROM support**).

XMM programs use the same program segment definitions as LMM programs - i.e:

- Code** : a read-only segment containing program code
- Cnst** : a read-only segment containing constant data
- Init** : a read/write segment containing static data
- Data** : a read/write segment containing dynamic data

Catalina provides an XMM Kernel for executing such programs, which knows how to make use of the external RAM. The Catalina XMM Kernel supports programs where the **Code** segment is located in (and executed from) external RAM but the other segments are located in Hub RAM (this is the **SMALL** memory model) as well programs where the **Code**, **Cnst**, **Init** and **Data** segments are all located in external RAM (this is the **LARGE** memory model). The Catalina XMM Loader arranges the segments in Hub RAM and XMM RAM before the program execution commences.

To load a Catalina XMM program, the XMM loader has to know where to get to the program segments from in the first place. One version of the XMM Loader supports loading from EEPROM – i.e. it knows how to move segments from EEPROM to HUB RAM, and from HUB RAM to XMM RAM. There is another version of the XMM Loader that supports loading from XMM to Hub RAM (this is used when loading an XMM program from SD Card, or via a serial connection to another Propeller).

For more details on XMM support for particular Propeller 1 platforms or add-on boards, check if there is a file called **platform_README.TXT** or **board_README.TXT** in the *target* folder. For example:

HYBRID_README.TXT

HYDRA_README.TXT

C3_README.TXT

HX512_README.TXT (for HYDRA and HYBRID)

FLASHPOINT_README.TXT (for RAMPAGE and SUPERQUAD)

XEPROM Support

Catalina can support executing XMM programs from EEPROM, on Propeller 1 platforms that have EEPROMS larger than 32kb, such as the HYDRA or the QUICKSTART.

In XEPROM mode, the code is executed from the EEPROM, and all data must be in Hub RAM. This means the SMALL memory model can be used.

A two-phase approach to loading XEPROM programs is used - the first 32 kb is loaded into Hub RAM first (as normal) but all this initial phase does is load and start the plugins, loads the data segments from the EEPROM into Hub RAM, then loads and starts the XMM Kernel which executes the XMM code (which remains in the EEPROM).

The files relevant to executing XMM code from EEPROM are in the target directory. They are:

XEPROM_XMM_DEF.inc	The pin and XMM Memory definitions for this board
XEPROM_XMM_CFG.inc	#defines to configure other Catalina plugins
XEPROM_XMM.inc	The XMM API code for this board

Also, there is a specific loader (*Catalina_XMM_XEPROM_Loader.spin*) used for loading and executing XEPROM programs, which is included automatically by the *xmm_default.spin* and *xmm_blackcat.spin* programs (although note that debugging programs executing from EEPROM is not supported yet).

Execution from EEPROM supports only the SMALL memory model, and it also requires that the cache be used. Execution from EEPROM is specified by defining the symbol XEPROM. Note that the XEPROM symbol must be specified IN ADDITION TO the symbol used for the base platform.

For example, a compilation command to use XEPROM on the QuickStart Platform (QUICKSTART) might look like:

```
catalina hello_world.c -lc -C QUICKSTART -C SMALL -C XEPROM -C CACHED
```

As usual, the Catalina utilities have to be compiled for the platform. This can be done using the **build_utilities** command. This will compile the EEPROM loader, which can then be used (in conjunction with payload) to load the XMM program into EEPROM. For example, after executing **build_utilities**, loading the above program might use a command such as:

```
payload EEPROM hello_world -i
```

Specifying the Memory Model

There are several Catalina command-line options that affect the layout of memory segments, the memory model, and the kernel to be used.

Early versions of Catalina used a single option (-x) to specify the memory layout, the memory model and the loader to be used:

- x memory layout, kernel addressing mode, and loader. This option is used to specify how the four program segments are arranged in memory, which kernel to use, the addressing mode the kernel should use, and also determine which loader must be used to load the program. While there are many possible arrangements of the four program segments, only a few currently have any use. These are:
 - x0 segments are arranged as **Code, Cnst, Init, Data** and the LMM kernel is used with the **TINY** addressing mode. This is the default mode.
 - x1 segments are arranged as **Code, Cnst, Init, Data** and the LMM kernel is used with the **TINY** addressing mode. This mode is used for EMM programs (see the section on **EMM Support** below).
 - x2 segments are arranged as **Cnst, Init, Data, Code** and the XMM Kernel is used with the **SMALL** address mode.
 - x3 segments are arranged as **Init, Data, Code, Cnst** and the XMM Kernel is used with the **LARGE** address mode. This mode is intended to be used when the Init and Data segments are to be placed in SPI RAM, and the **Code** and **Cnst** segment are to be placed in SPI Flash.
 - x4 segments are arranged as **Cnst, Init, Data, Code** and the XMM Kernel is used with the **SMALL** address mode. This mode is intended to be used when the **Code** segment is to be placed in SPI Flash.
 - x5 segments are arranged as **Code, Cnst, Init, Data** and the XMM Kernel is used with the **LARGE** address mode.
 - x6 segments are arranged as **Code, Cnst, Init, Data** and the LMM kernel is used with the **TINY** addressing mode. This mode is used for SMM programs (see the section on **SMM Support** below).
 - x8 segments are arranged as **Code, Cnst, Init, Data** and the CMM kernel is used with the **COMPACT** addressing mode. This mode is used for CMM programs (see the section on **CMM Support** above).
 - x9 segments are arranged as **Code, Cnst, Init, Data** and the CMM kernel is used with the **COMPACT** addressing mode. This mode is used for EMM programs (see the section on **EMM Support** below).
 - x10 segments are arranged as **Code, Cnst, Init, Data** and the CMM kernel is used with the **TINY** addressing mode. This mode is used for SMM programs (see the section on **SMM Support** below).

The Catalina Binder chooses a target (including the memory model and loader to use, and also the kernel to use) based on the selected layout.

The **-x** option is still supported, but the kernel and addressing mode can now be specified more easily by defining the symbols **COMPACT**, **TINY**, **SMALL** or **LARGE**, and the memory layout and loader to use can be specified by defining the symbols **SDCARD**, **FLASH** or **EEPROM** on the command line (using the **-C** command line option).

The meaning of each of these symbols is as follows:

TINY The default for Catalina is to produce TINY LMM programs. These programs can be up to 32 kb in size. This model corresponds to the normal Parallax mode used for SPIN or PASM programs.

TINY programs can be programmed into EEPROM, or loaded using *any* program loader, including the Parallax **Propeller Tool**, the Parallax **Propellant** loader, Catalina's **Catalyst** SD card loader, or Catalina's **Payload** serial loader.

SMALL On platforms that have external memory available, Catalina can produce SMALL XMM programs. These programs can have code segments up to 16 Mb in XMM RAM, but all data segments, the stack and the heap space must fit into the normal Propeller 32kb of Hub RAM.

SMALL programs can be loaded into EEPROM, or they can be loaded with Catalyst or Payload.

LARGE On platforms that have external memory available, Catalina can produce LARGE XMM programs. These programs can have code and data segments and heap space up to a total of 16 Mb in XMM RAM – the 32kb Hub RAM is used only for stack space.

LARGE programs can be loaded into EEPROM, or they can be loaded with Catalyst or Payload.

COMPACT Produce a CMM program. These programs can be up to 32 kb in size on the Propeller 1.

See the section **A Description of the Catalina Addressing Modes** later in this document for more details on the **TINY**, **COMPACT**, **SMALL** or **LARGE** addressing modes.

EEPROM TINY programs do not *need* any special processing to be executed from EEPROM, but there is a special Catalina target that allows them to be loaded in two phases, allowing such programs to make more effective use of the Propeller Hub RAM. For SMALL or LARGE programs to be executed from EEPROM, they *must* be compiled with a special Catalina target that knows how to load the XMM portion of the program from EEPROM to XMM RAM. The "special" Catalina targets are loaded into the *first* 32K of the EEPROM. The program is then loaded into the EEPROM starting at 32kb. The target loads all the plugins and drivers, and then loads the Catalina C program for execution – the advantage of using the special target (even for TINY programs) is that provided you have an EEPROM of at least 64 kb connected to your propeller, the C

program code does not need to share the same 32 kb space as the target code – allowing for larger C programs to be loaded and executed (see the section entitled **EMM Support** for more details).

SDCARD TINY, SMALL and LARGE programs do not *need* any special processing to be able to be loaded by Catalyst – but for TINY programs there is a special target that that allows them to be loaded in two phases, allowing such program to make more effective use of the Propeller Hub RAM. Like the special EEPROM loader, the SDCARD loader divides programs into several sections – the first 32kb contains the target, which sets up the execution environment for the program (by loading all the plugins and drivers etc). Then the *second* 32kb of the file contains the application program. Finally, the actual kernel is loaded from the last 2kb of the file. This means all program files compiled with the SDCARD loader option are exactly 66 kb in size (see the section entitled **SMM Support** for more details).

FLASH SMALL and LARGE programs that are to be executed out of SPI Flash RAM require a special load process, and so they must also use a special Catalina target. Both the Catalyst and the Payload loaders know how to load and run FLASH programs. Note that using FLASH requires the use of the caching SPI driver, so one of the cache options must also be specified (see the section titled **SPI Flash and Cache Support**).

The following command-line options can also affect the memory layout, and can be used with either method of specifying the memory layout (the **-x** method or the symbol definition method):

- M** XMM image size. This option is used to specify the maximum size of the resulting program image. If not specified, the default is 16 Mb.
- P** Read/Write Segment Address. This option is used in conjunction with the **-x3** and **-x4** options to specify the address to start the **Read/Write** segments (e.g. the data segment). This option is useful on platforms (such as the **C3**) which support the use of a combination of SPI RAM (read/write) and SPI Flash (read-only) as XMM RAM. Note that on all currently supported platforms that use SPI Flash, this is set automatically to the starting address of the SPI Flash, so there is usually no need to set this.
- R** Read-Only Segment Address. This option is used in conjunction with the **-x3** and **-x4** options to specify the address to start the **Read-Only** segments (e.g. the code segments). This option is useful on platforms (such as the **C3**) which support the use of a combination of SPI RAM (read/write) and SPI Flash (read-only) as XMM RAM. Note that on all currently supported platforms that use SPI RAM, this is set automatically to the starting address of the SPI RAM so there is usually no need to set this.

The following examples show both the **-x** method and the symbol definition method for specifying the memory mode/layout/kernel/loader. The following commands are equivalent:

```
catalina test.c -x1
catalina test.c -C EEPROM
catalina test.c -C TINY -C EEPROM
```

Note that in general you shouldn't mix the two methods – i.e. specifying **-x1** and **-C EEPROM** in the same command will result in a compilation error, since the symbol **EEPROM** will end up being defined twice.

Similarly, are following commands are equivalent:

```
catalina test.c -x6
catalina test.c -C SDCARD
catalina test.c -C TINY -C SDCARD
```

Again, note that you shouldn't mix the two methods – i.e. specifying **-x6** and **-C SDCARD** in the same command will result in a compilation error, since the symbol **SDCARD** will end up being defined twice.

Similarly, are following commands are equivalent:

```
catalina test.c -x4
catalina test.c -C SMALL -C FLASH
```

as are the following:

```
catalina test.c -x2
catalina test.c -C SMALL
```

and the following (note that **EEPROM** is supported by both methods when specifying XMM programs – this is a special case):

```
catalina test.c -x5 -C EEPROM
catalina test.c -C LARGE -C EEPROM
```

The following is a summary of the relationship between the two methods of specifying the memory modes, kernel and loader (you don't need to know this unless you need to migrate from using one method to using the other):

*The **-x0**, **-x1** and **-x6** layouts all use the **TINY** mode, and can be executed by the Catalina **LMM** Kernel. **-x10** uses the **COMPACT** mode and can be executed by the Catalina **CMM** Kernel. **-x0** and **-x1** layouts are supported all on platforms (with or without XMM hardware), and **-x6** and **-x10** are supported on all platforms with an SD card. **-x1** means use the **EEPROM** loader, and **-x6** and **-x10** means to use the **SDCARD** loader.*

*The **-x2** layout uses the **SMALL** memory model, and the **-x5** layout uses the **LARGE** memory model. These layouts can only be executed by the Catalina **XMM** Kernel, and require platforms with XMM hardware. These layouts require **Read/Write** XMM RAM, and use the XMM LMM Loader unless **EEPROM** is defined, in which case they use the XMM **EEPROM** Loader.*

*The **-x3**, layout uses the **LARGE** mode, and the **-x4** layout uses the **SMALL** mode. These layouts can only be executed by the Catalina **XMM** Kernel, and require platforms with XMM hardware. These layouts are intended for platforms with Read-Only XMM RAM, such as the SPI Flash on the C3, SuperQuad, RamPage or RamPage 2. These layouts use the **FLASH** Loader,*

*and are equivalent to defining the symbol **FLASH** in conjunction with the **LARGE** and **SMALL** symbol. The Flash XMM Loader is used in these cases.*

EMM Support

EMM programs are similar to TINY LMM or COMPACT CMM programs, but use a special EMM Loader. The EMM Loader is designed to load **TINY** LMM programs from above address \$8000 in an external EEPROM into Hub RAM. Other than this, EMM is very similar to LMM, and uses the same Kernel. EMM programs use their own set of target files.

*Note that there is also a special EEPROM loader option for **SMALL** and **LARGE** XMM programs – this is not quite the same as an EMM program even though both are enabled by defining the **EEPROM** symbol – if the program is a TINY LMM program this means it will use the *emm* target and the LMM EEPROM loader, whereas if the program is an XMM program it uses the *xmm* target and the XMM EEPROM loader option.*

To understand the advantage of compiling C programs as EMM programs, consider the *hello_world.c* program. When compiled using the default LMM target (and using the *libci* library), this simple program occupies around 15 kb of Hub RAM. Most of this RAM space is occupied by the C library code required to support the program, but a significant portion is also occupied by the LMM target and various plugins required to run it. Even so, this would appear to make the Propeller almost unusable for real C programs. However, when the same program is compiled as an EMM program, the C program code occupies only 7 kb – this means that if we compile it as an EMM program, then even after loading the stdio **printf** functions (which are quite memory hungry) there is still another 25 kb available for C programs.

In summary, the advantage of EMM over the normal LMM mode is that larger C programs can be constructed – even without requiring XMM RAM. While the program can still only be 32k, it does not have to share that 32k space with the target and plugin code. However, EMM programs require an external EEPROM of 64 kb (or larger).

To request Catalina use the EMM loader for a TINY LMM program, specify the **-x1** option to the Catalina Compiler or Binder, or define the symbol **EEPROM** on the command line when compiling a normal TINY LMM program. When this option is specified, the Catalina Binder uses a new set of targets, prefixed by *emm_*. Currently one default EMM target (called *emm_default*) and one debug target, (called *emm_debug*) are provided.

To request Catalina use the EMM loader for a COMPACT CMM program, specify the **-x9** option to the Catalina Compiler or Binder, or define the symbol **EEPROM** on the command line when compiling a normal COMPACT CMM program. When this option is specified, the Catalina Binder uses a new set of targets, prefixed by *emm_*. Currently one default EMM target (called *emm_default*) and one debug target, (called *emm_debug*) are provided.

*Note that XMM programs can also be compiled with the symbol **EEPROM** defined. This uses the normal xmm targets (called `xmm_default` and `xmm_debug`) but specifies the XMM EEPROM loader option be used.*

Further advantages for EMM mode are expected to be achieved in subsequent Catalina releases, due to the replacement of some of the basic C library functions with ‘plugin’ based equivalents – now that the EMM loader is available, these functions will no longer need to consume Hub RAM at run time.

SMM Support

SMM programs are similar to TINY LMM or COMPACT CMM programs, but use a special SMM Loader. The SMM Loader is designed to load programs from an SD Card Hub RAM. Other than this, SMM is very similar to LMM and CMM, and uses the same Kernels.

The advantage of compiling C programs as SMM programs is similar to the advantage of compiling them as EMM programs – i.e. the SMM Loader is a two-phase loader, which allows more of the Hub RAM (up to 31kb) to be used for C program code.

SMM programs must be loaded from an SD Card, using the **Catalyst** program loader.

To request Catalina use the SMM loader, specify the **-x6** or **-x10** option to the Catalina Compiler or Binder, or define the symbol **SDCARD** on the command line. When this option is specified, the Catalina Binder uses a new set of targets, prefixed by `smm_`.

Currently one default SMM target (called `smm_default`) and one debug target, (called `smm_debug`) are provided.

Catalina Cog Usage

The number of cogs used by a Catalina program depends on the target, and the plugins and drivers loaded by that target.

To figure out how many cogs are required, the following table is provided:

Plugin/Driver Name	Cogs
Any HMI plugin	+ 1
LoRes TV driver	+ 1
HiRes TV driver	+ 1
LoRes VGA driver	+ 1
HiRes VGA driver	+ 2
Mouse driver	+ 1
Keyboard driver	+ 1
Float32_A	+ 1
Float32_B (and Float32_A)	+ 2

Real-Time Clock	+ 0 / + 1 ¹⁹
Random plugin	+ 1
SD Card	+ 1
Caching SPI XMM driver	+ 1
Gamepad plugin	+ 1
Sound plugin	+ 1
SPI plugin	+ 1
CGI Graphics plugin	+ 2
VGI Graphics plugin	+ 5
Kernel (CMM, LMM, or XMM)	+ 1

Examination of the table above will show that it is relatively easy to specify options to the standard targets that would exceed the 8 cog limit – doing so will cause one or more plugins or drivers to fail to load – and most likely “hang” the C program when it attempts to use the plugin that failed to load.

The Kernel cog is *always* used to execute Catalina C code, but Catalina can also run C code on any cog not used for other purposes – for more details, see the **Multi-Cog Support** section (below).

It is of course possible to create new dedicated targets that load different drivers, such as a combined keyboard/mouse driver that takes only one cog. The standard set of targets, drivers and plugins provided are intended to be functionally rich, but they do not necessarily make the most efficient use of the available cogs.

Supporting multiple Propeller platforms

A single target package can provide support for multiple Catalina platforms. This is done by using conditional compilation in the various target files.

To support a new platform there are two options:

- Create a new target directory specifically for the new platform.
- Add the new platform into the existing target directory by including appropriate conditionally compiled sections to the existing target files.

In practice, it may be best to do both – i.e. to start out with a copy of the existing target files in a new target directory, modifying them as required to get the new platform working – and then integrate the results into the standard target directory using appropriate conditional compilation flags.

The Catalina compiler expects the default target package to be called *target*, but a target directory can be called anything, and referenced by using the **-T** option to Catalina.

Compiling for a platform supported in the standard target directory uses commands such as:

¹⁹ The Real-Time Clock only uses an extra cog if the SD Card plugin is not enabled – if both the SD and CLOCK plugins are both loaded, then the SD plugin is used for both functions, so the clock does not occupy an extra cog.

```
catalina hello_world.c -lc
```

When compiling for a platform supported in a different target directory, the equivalent command would be something like:

```
catalina hello_world.c -lc -T"C:\Program Files (x86)\Catalina\My_Target"
```

Target Packages

The standard target package (target)

The **default** Target Package (in subdirectory *target*) can be used with any of the supported Propeller **base** platforms (i.e. **HYDRA**, **HYBRID**, **C3** etc). It also includes the default **CUSTOM** platform, which is suitable for nearly any Propeller equipped with a 5Mhz crystal.

The default target platform also supports XMM add-on cards, such as the **RAMPAGE** and **RAMPAGE2**, **SUPERQUAD** or **PMC** (Propeller Memory Card),

Note that the user configurable parts are separated out into “include” files, depending on whether the files are for a base platform board, or an add-on board:

For a base platform (XXX) that has no XMM RAM, there are three files:

<i>XXX_DEF.inc</i>	general pin and clock definitions for the platform
<i>XXX_CFG.inc</i>	plugin configuration options for the platform
<i>XXX_HMI.inc</i>	HMI options supported by the platform

For an XMM RAM add-on board (YYY), there are three different files:

<i>YYY_XMM.inc</i>	XMM API functions for the add-on board
<i>YYY_XMM_DEF.inc</i>	XMM pin and memory definitions for the add-on board
<i>YYY_XMM_CFG.inc</i>	XMM plugin configuration options for the add-on board

For a base platform (ZZZ) that also has built-in XMM RAM, there will be six files:

<i>ZZZ_DEF.inc</i>	general pin and clock definitions for the platform
<i>ZZZ_CFG.inc</i>	plugin configuration options for the platform
<i>ZZZ_HMI.inc</i>	HMI options supported by the platform
<i>ZZZ_XMM.inc</i>	XMM API functions
<i>ZZZ_XMM_DEF.inc</i>	XMM pin and memory definitions
<i>ZZZ_XMM_CFG.inc</i>	XMM plugin configuration options

This package includes targets for LMM and XMM programs. It includes support for the alternate, threaded and dynamically loadable LMM kernels.

This package supports the normal Parallax Serial loader, as well as the **SDCARD**, **EEPROM** and **FLASH** program loaders.

This package supports the BlackBox debugger, plus the POD debugger.

This package supports all Catalina plugins. For each target, it automatically includes the SD card and floating point plugins as required to support the library options selected, as well as various other plugins that can be manually specified (such as the Clock or various HMI plugins).

This target package is intended for general-purpose use.

The embedded target package

The **embedded** Catalina Target Package (in sub-directory *embedded*) provides support for only *one single* Propeller platform. However, the user configurable parts are separated out into the following “include” files for convenience:

Custom_DEF.inc general pin and clock definitions for the platform
Custom_CFG.inc plugin configuration options for the platform
Custom_XMM.inc XMM support functions for the platform (if applicable)

This package includes targets for LMM and XMM programs. It includes support for the alternate, threaded and dynamically loadable LMM kernels.

This package supports the normal Parallax Serial loader, as well as the SDCARD, EEPROM and FLASH program loaders.

This package supports the BlackBox debuggers, but not the POD debugger.

This package supports a subset of the Catalina plugins. For each target, it automatically includes the SD card and floating point plugins as required to support the library options selected, but no other plugins (specifically, it does not support any HMI plugins for keyboard, screen or mouse support). If these or other plugins are required, they can be added manually.

This target package is intended for users to create a dedicated target for a specific application or platform.

The minimal target package

The **minimal** Catalina Target Package (in subdirectory *minimal*) is an intentionally “minimalist” target package. It provides support for only *one single* Propeller platform. However, the user configurable parts are separated out into the following “include” file for convenience:

Custom_DEF.inc general pin and clock definitions for the platform
Custom_CFG.inc plugin configuration options for the platform

This package supports only LMM programs.

This package supports only the normal Parallax Serial loader.

This package does not support any debuggers.

This package supports only one custom plugin.

The purpose of this package is primarily educational. For more information on its use, refer to the document **Getting Started with Plugins**.

Using PASM with Catalina

The Catalina compiler conforms to the ANSI C standard. This standard does not define a specific keyword (or function, or technique) for the inclusion of code written in assembly language in a C program.

However, it is possible to incorporate PASM assembly language code into Catalina C programs, using at least four different techniques:

1. Using the **PASM** function to include PASM instructions 'inline' with C code.
2. Write a target that loads the PASM program during initialization.
3. Convert the PASM program into a Catalina *plugin* and load it during initialization (as is done for the various HMI drivers, the floating point libraries, and the SD card and clock drivers).
4. Load the compiled binary version of a PASM program into a spare cog from within the C program (using the **_coginit** function).
5. Write a subroutine in LMM PASM and call it from the C program in the same way that any C function is called.

Each of these techniques is described in more detail below, after a brief description of the various different types of PASM.

The PASM must be specially written to allow it to be executed by the Kernel, and Catalina has several different Kernels:

LMM PASM (i.e. PASM intended to be executed by the LMM or XMM Kernel)

CMM PASM ((i.e. PASM intended to be executed by the LMM Kernel

pure PASM (i.e. PASM intended to be executed directly by a cog).

While many LMM or CMM instructions are identical to pure PASM, some pure PASM instructions cannot be executed within the kernel. Instead, they must be replaced by kernel equivalents known as *primitives*.

A good example of this is the PASM **jmp** instruction. If this instruction were executed within the LMM kernel, the program would jump to the corresponding location within the kernel itself, not the desired location in the PASM program. So Instead of using **jmp**, a new LMM PASM primitive (called **JMPA**) is provided.

In pure PASM, a **jmp** instruction might look as follows:

```
loop jmp #loop      ' loop forever
```

In LMM PASM, this would have to be replaced by the following:

```
loop jmp #JMPA      ' loop ...
      long @loop     ' ... forever
```

More information on the pure PASM instructions that need to be replaced by LMM PASM primitives are given in **A Description of the Catalina Virtual Machine** (later in this document).

Inline PASM

The PASM function

Catalina defines a **PASM** function that can be used for including PASM instructions inline with C code. The prototype for this function (defined in *propeller.h*) is:

```
extern void PASM(const char *code);
```

However, there is no actual PASM function body. Instead, when it sees a call to this function, the compiler inserts the *string literal* argument `code` (it cannot be a variable) into the assembly language output, to be assembled along with the PASM generated by the Compiler..

For example, the following program will toggle the Propeller's P0 output every 500 milliseconds:

```
void main() {
    PASM(" or dira, #1");      // set bit 0 as output
    while(1) {
        _waitms(500);
        PASM(" xor outa, #1"); // toggle bit 0
    }
}
```

Note that multiple PASM statements can be inserted. This can be done in several ways. The following are all equivalent:

1. With multiple successive PASM functions:

```
PASM(" or dira, #1");
PASM(" xor outa, #1");
```

2. With multiple statements in one string to the PASM function, separated by new line characters::

```
PASM(" or dira, #1\n xor outa, #1");
```

3. With multiple statements in multiple strings to the PASM function - there is nothing special about the PASM function - in C multiple strings can generally appear wherever one string can appear, and they are simply concatenated. Note that each string except the last should terminate with a new line character:

```
PASM(
    " or dira, #1\n"
    " xor outa, #1"
);
```

The _PASM macro

Catalina defines a **_PASM** macro that can be used in the PASM function to determine the PASM name the compiler assigns to a C identifier. Note that this is not

a general-purpose C macro - it can only be used within the string literal of a PASM function.

The **_PASM(name)** macro can be used for:

global variables	returns the PASM label of the C variable name .
functions	returns the PASM label of the C function name .
function arguments	returns the register or frame offset of the function argument name .

If the argument to the **_PASM** macro is not recognized as the name of any of the above, the name is simply returned - e.g. the result of **_PASM(function_name)** might be **C_function_name** if there is a C function defined with the name, or just **function_name** if there is not.

Note that **_PASM()** does NOT work for local variables. This is because at the time of macro expansion, the final location of local variables is not yet known - some may even end up not being allocated at all (e.g. if they are not used in C, or are used only in PASM but not in C - this is because the C compiler does not know how to interpret PASM string literals. But note that you can *force* the location of local variables to be fixed at a specific point by making them arguments to a function - even if that function is subsequently 'inlined'.

For example:

```
int sum(int a, int b, int c, int d) {
    return PASM(
        " mov r0, _PASM(a)\n"
        " add r0, _PASM(b)\n"
        " add r0, _PASM(c)\n"
        " add r0, _PASM(d)\n"
    );
}

void main() {
    int a = 1;
    int b = 2;
    int c = 3;
    int d = 4;
    int total;

    // the following will NOT work because the _PASM
    // macro cannot be used on local identifiers ...
    /*
    total = PASM(
        " mov r0, _PASM(a)\n"
        " add r0, _PASM(b)\n"
        " add r0, _PASM(c)\n"
        " add r0, _PASM(d)\n"
    );
    */
}
```

```

// however, the following WILL work - and the
// overhead of the function call can be eliminated
// by using the Catalina optimizer ...
total = sum(a, b, c, d);
printf("total = %d\n", total);
}

```

Note that there are constraints on the PASM that can be used in conjunction with C via this technique. For example, if the program is compiled in TINY mode, both the C code and the 'inline' PASM is executed within an LMM virtual machine. For details on these constraints, see the section later in this document entitled ***A Description of the Catalina Virtual Machine.***

Also note that the PASM differs between the Propeller 1 and Propeller 2, and also between NATIVE, TINY and COMPACT modes. Use **#ifdef** to incorporate different PASM if required.

For example, the following code demonstrates adding an integer to a global variable, using the correct PASM when compiled either in NATIVE mode or TINY mode:

```

static value;

void add_to_value(int a) {
    PASM(
        "#ifdef NATIVE\n"                // propeller 2 NATIVE PASM
        " rdlong r0, ##@_PASM(value)\n"
        " add r0, _PASM(a)\n"
        " wrlong r0, ##@_PASM(value)\n"
        "#else\n"                        // propeller 1 or 2 TINY PASM
        " jmp #LODL\n"
        " long @_PASM(value)\n"
        " rdlong r0, RI\n"
        " add r0, _PASM(a)\n"
        " wrlong r0, RI\n"
        "#endif\n"
    );
}

```

It has been mentioned that the Catalina C compiler cannot interpret the PASM strings, and this means that it is unable to take into account whether the PASM is used by the program or not. This can lead to problems when the Catalina Optimizer is used, because that may eliminate code that it believes is not required by the C program - but the code so eliminated may contain inline PASM that *is* required. In such cases, simply declare a reference to the function that you do not want the Optimizer to remove, so that it thinks the function is used.

For example, this is demonstrated in the following code::

```

// this function exists only to define storage for
// long abc - it cannot actually be called from C ...
void abc(void) {
    PASM("abc long 12345\n");
}

```

```

// this function returns the value of abc ...
int read_abc() {
    return PASM(
        "#ifdef NATIVE\n"           // propeller 2 NATIVE PASM
        " rldlong r0, ##@abc\n"
        "#else\n"                   // propeller 1 or 2 TINY PASM
        " jmp #LODL\n"
        " long abc\n"
        " rldlong r0, RI\n"
        "#endif\n"
    );
}

// this type is required to declare a pointer to abc ...
typedef void (*dummy_t)(void);

void main() {

    // without the following line, the function abc would be
    // removed by the Catalina Optimizer, because it looks
    // like it is unused by the C program (it is used, but
    // only from within the inline PASM in read_abc) ...
    dummy_t dummy = abc;

    printf("value of abc is %d\n", read_abc());
}

```

For more details and more examples of inline PASM, examine the various *test_inline_pasm_n.c* files in the *demos\inline_pasm* folder.

The **_PSTR** macro

Catalina defines a **_PSTR** macro that can be used in the PASM function to convert a C string to a PASM string, decoding any C escape sequences embedded in the C string. Note that this is not a general-purpose C macro - it can only be used within the string literal of a PASM function.

_PSTR(string) produces a PASM string from its C string argument. It is intended to allow C strings to be used within PASM functions the same way they would be used in C. The arguments to the PASM function - including any embedded strings - are always interpreted as PASM strings, and PASM does not interpret C escape sequences (such as `\n`) embedded in the string. But **_PSTR()** can be used to do the same thing.

Suppose for instance that we want to use the C string `"hello\n"` in a PASM statement. That is, the characters `h e l l o` followed by a **newline**.

First, we might try:

```
PASM("byte \"hello\"\n");    // this will not compile
```

Then we might try:

```
PASM("byte \"hello\\n\"); // this compiles, but the \n
                          // is not encoded as 0x0a, it
                          // is encoded as 0x5c 0x6e
```

The only way to get the desired result is to interpret the embedded C escape sequence manually, such as:

```
PASM(" byte \"hello\\n byte $0a\n");
```

But this gets difficult if there are many C escape sequences embedded in the string. Fortunately, there is a simpler method - use **_PSTR()** on any C strings embedded in the argument to PASM:

```
PASM("_PSTR(hello\n)"); // compiles correctly - \n is
                        // encoded as 0x0a
```

or

```
PASM("_PSTR(\"hello\\n\")"); // compiles correctly
```

_PSTR() interprets the following C escape sequences:

\a	07	Alert (Beep, Bell)
\b	08	Backspace
\f	0C	Form Feed Page Break
\n	0A	Newline (Line Feed); see notes below
\r	0D	Carriage Return
\t	09	Horizontal Tab
\v	0B	Vertical Tab
\\	5C	Backslash
\'	27	Apostrophe or single quotation mark
\"	22	Double quotation mark
\?	3F	Question mark (used to avoid trigraphs)
\nnn	any	The byte whose numerical value is given by nnn interpreted as an octal number
\xhh...	any	The byte whose numerical value is given by hh... interpreted as a hexadecimal number

Notes:

\a .. \? all produce one byte.

An octal escape sequence consists of **** followed by one, two, or three octal digits. The octal escape sequence ends when it either contains three octal digits already, or the next character is not an octal digit. For example, **\11** is a single octal escape sequence denoting a byte with numerical value 9 (11 in octal), rather than the escape sequence **\1** followed by the digit **1**.

However, `\1111` is the octal escape sequence `\111` followed by the digit `1`. Note that some three-digit octal escape sequences may be too large to fit in a single byte; this results in the value being masked to 8 bits, so `\777` gives the same value as `\377`.

A hexadecimal escape sequence must have at least one hex digit following `\x`, with no upper bound on the number of hex digits; it continues for as many hex digits as there are following the `\x`. Thus, for example, `\xABCDEFG` denotes the hexadecimal value `ABCDEF`, followed by the letter `G` (because `G` is not a hex digit). If the hex value is too large to fit in a single byte it is masked to 8 bits. For example:

`\x12` single byte with hex value `0x12` (i.e. 18 decimal)

`\x1234` single byte with hex value masked to 8 bits (i.e. `0x34`)

The argument to `_PSTR()` is a string. It can be quoted or not if used directly in a PASM statement. This means a string like `_PSTR("HELLO")` is the same as `_PSTR(HELLO)` when used within a PASM statement, but if used within a C string it would need to be expressed as `_PSTR("HELLO")`.

`_PSTR()` generates a sequence of PASM byte statements to encode the string, but unlike C it does not null terminate it. This allows for string concatenation. For example:

```
_PSTR(DEAD) _PSTR(BEEF)
```

is the same as

```
_PSTR(DEADBEEF)
```

The escape sequence `\0` is a commonly used octal escape sequence, which denotes the null character (i.e. with value zero). The same thing would be achieved by the hexadecimal escape sequence `\x0`. However, using either of these can result in a null byte in the middle of a string, and since the string processing is done in C and C treats a null as a string terminator, this will cause the string to be truncated, which is probably not what was intended. To terminate a string, add an explicit **byte 0\n** after the call to `_PSTR()` and to insert a null character in a string use two calls to `_PSTR()` with an explicit **byte 0\n** in between. For example:

```
_PSTR(DEAD) byte 0\n_PSTR(BEEF) byte 0\n
```

This would result in:

```
byte $44
byte $45
byte $41
byte $44
byte 0
byte $42
byte $45
byte $45
byte $46
byte 0
```

If a `\` appears as the last character in a C string (i.e. followed by a null terminator) it is treated as a normal character, not as an escape sequence.

Load the PASM program at initialization time

Each Catalina target is a normal SPIN program whose job is to establish the execution environment for the Catalina C program. However, this program can execute any PASM or SPIN code, including loading PASM programs into cogs to be run in parallel with the C program. Of course, the PASM program must not read or write to Hub RAM except under well defined circumstances - e.g. by only writing to an area of high RAM that Catalina reserves for this purpose.

For an example of this technique, see the *Catalina_Cogstore.spin* program.

This is a normal PASM program started by various Catalina targets to assist in decoding command-line arguments passed to the program. In this particular case the PASM program is stopped again once its work is completed – but it could be left running if necessary. Examples of the latter include the various debugger targets (e.g. *Imm_blackcat.spin*).

This technique is not discussed any further in this document.

Convert the PASM program into a Catalina plugin

Plugins are a very versatile solution since they can interact with the Catalina C program at run time - but they can be complex to develop and can also be expensive in resources (since they cannot be loaded and unloaded on demand - they are expected to be loaded once and then remain running for the duration of the C program).

However, plugins are the best solution when the PASM program and the C program are required to interact since there is a well-defined interface that supports communication between a C program and any plugins that have been loaded to support it.

There are many examples provided in the Catalina *target* directory. Most standard Parallax drivers can be easily converted into plugins.

This technique is not discussed any further in this document.

Load a compiled PASM program into a cog

Catalina provides a `_coginit` function that works in a very similar manner to the corresponding SPIN or PASM `coginit` operations - i.e. it is used to load a binary PASM program into a cog for execution.

A tool to assist in converting a PASM binary into a form suitable for loading from C is provided - this tool is called **spinc**. It is provided in both source and executable form. Thanks go to Steve Densen for developing the original version of this useful tool!

A specific example of using **spinc** is provided, called *test_spinc.c*. To build it, use the **build_all** batch file provided. E.g:

```
build_all HYDRA
```

This batch file does the following commands:

- a) compiles the *flash_led.spin* PASM program (to produce *flash_led.binary*):

```
spinnaker -p -a flash_led.spin -b
```

- b) converts the *flash_led.binary* to a C include file:

```
spinc flash_led.binary > flash_led_array.h
```

- c) compiles a C program which loads the resulting binary using **_coginit**:

```
catalina -lc -I. test_spinc.c
```

Examine each of the files mentioned above for more detail.

NOTE: Even though you specify your platform when building, you may still need to modify the clock speed and pin numbers defined in the file *flash_led.spin* to make it work correctly on your platform.

To load and execute the resulting program, simply type:

```
payload test_spinc
```

Writing an LMM PASM function that can be called directly from C

A Catalina C program can call a PASM function directly. However, the PASM must be specially written to allow it to be executed by the LMM Kernel.

LMM PASM (i.e. PASM intended to be executed by the LMM Kernel) is slightly different to *pure PASM* (i.e. PASM intended to be executed directly by a cog). While many LMM PASM instructions are identical to pure PASM, some pure PASM instructions cannot be executed within the kernel. Instead, they must be replaced by LMM equivalents known as *primitives*.

A good example of this is the PASM **jmp** instruction. If this instruction were executed within the LMM kernel, the program would jump to the corresponding location within the kernel itself, not the desired location in the PASM program. So Instead of using **jmp**, a new LMM PASM primitive (called **JMPA**) is provided.

In pure PASM, a **jmp** instruction might look as follows:

```
loop jmp #loop      ' loop forever
```

In LMM PASM, this would have to be replaced by the following:

```
loop jmp #JMPA      ' loop ...
long @loop          ' ... forever
```

More information on the pure PASM instructions that need to be replaced by LMM PASM primitives are given in **A Description of the Catalina Virtual Machine** (later in this document).

A full working example of this technique is provided in the *demos/spinc* directory. It is called **test_pasm**. To build it, use the **build_all** batch file provided - e.g:

```
build_all HYDRA
```

This batch file executes the following command:

```
catalina -lc test_pasm.c flash_led.obj
```

The file *flash_led.obj* is the LMM PASM program - giving it the **.obj** extension tells Catalina that it is not a C file that needs to be compiled - it is a PASM file that is ready to be bound. In fact, all Catalina **obj** files are LMM PASM programs and can be viewed with any text editor. Examine both *flash_led.obj* and *test_pasm.c* for more details.

NOTE: Even though you specify your platform when building, you may still need to modify the clock speed and pin numbers defined in the file *flash_led.obj* to make it work correctly on your platform.

To load and execute the resulting program, simply type:

```
payload test_pasm
```

Precautions when using LMM PASM with the Catalina Optimizer

If you plan to write LMM PASM functions that are called from C, and also use the Catalina Optimizer, be aware that there are some additional precautions that must be taken in the hand-written LMM PASM. This is because of the *inlining* function of the optimizer:

1. Every subroutine must contain *exactly one*, **RETN** or **RETF** instruction, and it must be at the *end* of the function. If you need to exit earlier, instead use a jump to the common exit point at the end of the function. This technique will be familiar to most PASM programmers because of the way the PASM CALL instruction works) For example:

```
my_pasm_routine
    cmp r0, #0 wz          \ if r0 = 0
    jmp #BR_Z              \ ... then ...
    long @my_common_exit   \ ... exit
    sub r0, #1             \ otherwise decrement r0
my_common_exit
    jmp #RETN
```

2. Do not use “local” symbol names (i.e. symbols that start with ‘:’) within an LMM PASM function unless you are sure that the resulting code will not end up with two identical local symbol names within the same function if the function is inlined. If in doubt, simply avoid local symbols altogether and use global symbols instead.

Multi-CPU Support

Catalina fully supports all CPUs in multi-CPU systems. Currently, there is only such systems supported by Catalina:

TriBladeProp – this platform has 3 ‘blades’, each with one CPU:

- CPU #1 can have XMM RAM and keyboard, mouse and screen (TV or VGA) devices attached.
- CPU #2 can have XMM RAM and an SD card attached.
- CPU #3 has no specific devices attached, and is intended for adding other I/O.

CPU #2 can communicate serially with (and load programs into, or reset) CPU #1 and CPU #3.

Note that CPU #1 and CPU #2 use slightly different implementations for their XMM RAM.

Multi-CPU systems can be complex to configure – not least of which because the normal set of devices is often distributed amongst the various CPUs. Another complicating factor is that not all CPUs may have boot EEPROMs attached – they may be expecting to be loaded from another CPU.

Catalina supports multi-CPU systems in two ways:

1. Providing a set of drivers for “proxy” devices. These drivers allow programs running on one CPU to have access to a device (HMI or SD) physically connected to another CPU. The client CPU runs the proxy drivers and the server CPU (i.e. the one with the actual physical devices) runs a special proxy server program which services device requests from the client. The communication between the client and the server uses serial communications between the Props. The proxy drivers are as close as possible in functionality to the ordinary drivers, and can usually be used without any C code changes. The main difference is that their performance will typically be slower because the proxy driver has to communicate serially with the real driver running on the other CPU to perform each driver function.
2. Providing a set of utilities that allows one CPU to control another CPU – i.e. to reset or load programs into that CPU. Programs can be loaded into RAM or EEPROM (if the target CPU has a boot EEPROM attached).

Proxy Devices

Using proxy devices always requires the execution of a server program on the CPU that is physically connected to the actual devices. The devices that can be “proxied” are:

The SD Card

The Screen

The Keyboard

The Mouse

Generic_Proxy_Server

A **Generic_Proxy_Server** program is provided that can be reconfigured to suit any combination of proxy devices. Only one instance of the program needs to be executed on a CPU – it acts as proxy for all the devices for which it is configured.

To use the proxy devices from a client CPU, all the normal Catalina command line options should be specified (i.e. as if all the devices were local) but there are additional options that can be specified to indicate that some devices are not local, but will be provided via a proxy server:

```
PROXY_SD
PROXY_SCREEN
PROXY_MOUSE
PROXY_KEYBOARD
```

For example, suppose we have a multi-prop system where CPU #1 has a keyboard and mouse (but no screen or SD card), and CPU #2 has a TV output and an SD Card (but no keyboard or mouse).

Then we might use commands similar to the following to compile a program intended to run on CPU #1:

```
catalina prog_1.c -lcx -C CPU_1 -C TV -C PROXY_SD -C PROXY_SCREEN
```

This command tells Catalina to compile *prog_1.c* for execution on CPU #1 using real drivers to access the local keyboard and mouse, but to use proxy drivers to access the SD card and screen (since they are on another CPU).

On CPU #2 we then run an instance of the **Generic_Proxy_Server** program, configured to act as proxy for the screen (using the TV output) and the SD card devices.

However, the **Generic_Proxy_Server** program is not a Catalina program – it is a SPIN/PASM program that is compiled using **spinnaker**:

```
spinnaker -p -a Generic_Proxy_Server -o my_proxy -I..\target -b -C
CPU_2 -C TV -C NO_KEYBOARD -C NO_MOUSE
```

These commands tell **spinnaker** to generate a proxy server binary (called *my_proxy.binary*) suitable for use on CPU #2. It will include a TV screen driver and an SD card driver, but no keyboard and no mouse driver (since those devices are not present on this CPU).

The only ‘special’ option required when configuring the proxy server is a new **NO_SD** option, which must be specified to exclude the SD card driver on CPUs that do not have this device (there is no equivalent Catalina command line option – the SD driver is always included if you link with the **-lcx** library versus the **-lc** library). Other than this, the configuration options used to configure the proxy server are the same as those used for Catalina programs (e.g. in the above case we used the **TV** and **CPU_2** options).

There are normally only one or two different proxy server configurations required in a multi-CPU system, so the proxy server does not usually need to be recompiled for each client –the same proxy server binary may be used for many clients. In some cases, it is worthwhile including actual drivers in the proxy server even though they are not often used. For example, if one CPU in a multi-CPU system only has an SD card, we may choose to configure a proxy server for that CPU that also includes PC drivers – that way, the same proxy could be used by any program requiring either a PC connection, or just the SD Card.

Proxying **PC** and **PROPTERMINAL** devices is also supported – but this at first may appear a little confusing, since these devices are already a type of proxy device (i.e. the actual devices are not directly connected to the CPU that uses them). When you proxy a PC or PROPTERMINAL device from one CPU to another, what actually happens is that serial request from the client to the PC (e.g. to output a character) is proxied from the client CPU to the server CPU, and the server CPU then forwards the request to the PC (and vice versa for the response). The most common reason for re-proxying these devices is that by doing so you only need to have one serial connection from the PC to the multi-CPU system – i.e. a proxy running on the CPU with the serial connection to the PC can be used to allow another CPU in the multi-CPU system to communicate with the PC as if it had its own serial connection. This is particularly useful in systems where the CPUs use serial pins other than the normal Propeller SI and SO pins (which would otherwise necessitate special hardware to communicate with the PC).

There are so many possible ways to proxy devices even in a simple 2-CPU system that the best way to understand proxy devices is to try out a few cases – see the document **Getting Started with Catalina** for a tutorial containing some examples.

Note that the behavior of programs that use proxy devices is undefined if the configuration specified for the proxy server does not match the configuration specified for the client programs – e.g. in the example given above, if the **Generic_Proxy_Server** program was actually connected to a local **VGA** display, but the client was compiled expecting to use a **TV** display, then some of the screen features may work while others may not – the results are unpredictable as they depends on the capabilities of each of the drivers involved.

Resetting and/or Loading another Prop

Since some CPUs in a multi-CPU system may not even have EEPROMs installed, utility programs are provided that can be used to load a program into either the RAM or EEPROM of another CPU, or to remotely reset another CPU. These programs take advantage of the built-in boot load capabilities of the Propeller CPU, which are sufficient to allow one blade to load a small SPIN (or LMM program) into another CPU. However such utilities can only load programs that have been previously embedded into the loader program itself, and can only load programs into Hub RAM - to load programs from the SD card, or to load programs into XMM RAM requires a more sophisticated loader to be running on the 'master' CPU, and also requires a companion loader program to be running on the 'slave' (destination) CPU.

Catalina therefore provides the Catalyst SD Loader program. Catalyst can be used in conjunction with an SD Card adaptor to load either SPIN programs, or Catalina LMM or XMM programs from the SD Card to the local CPU. It can also load programs via serial I/O to other CPUs. Programs that fit into 31k (such as SPIN programs or Catalina LMM programs) require no special handling to be loaded using this Loader, but Catalina XMM programs have to be compiled differently depending on whether they are intended to be loaded from the SD card – either directly or via Serial I/O to another CPU – or from an EEPROM. The default is to compile programs ready to be loaded from the SD Card or serial I/O. Loading from EEPROM requires the -C EEPROM command line option to be specified.

To support the serial capabilities of the Catalyst Loader, a Generic SIO Loader program is also provided – this is a 'companion' loader that must be run on the destination CPU to support the serial load process. There are several ways to accomplish this – if the destination CPU has an EEPROM installed, the Generic SIO Loader can be programmed into the EEPROM to always start on boot. If not, the Generic SIO Loader can be downloaded from the PC. But the usual method if the destination CPU has no EEPROM is to first use the Catalyst Loader program to load a special program that has the Generic SIO Loader embedded within it using the Propeller's built in boot capabilities. Once the companion loader is running, any other program can be downloaded to the CPU.

Catalina provides the following utility programs to support loading programs into various CPUs (in the *Catalina\utilities* sub-directory):

Catalina_XMM_SD_Loader

This program is used by the Catalyst SD Program Loader. Catalyst is designed to run on a CPU with direct access to an SD card, and allows SPIN, LMM or XMM programs to be loaded from the SD Card into either the local CPU, or into another CPU. This program is a normal SPIN/PASM program, but it can use any of the Catalina HMI plugins for the user interface – including being used remotely using a PC terminal emulator such as the Parallax Serial Terminal.

Catalyst knows how to load SPIN, LMM or XMM programs. SPIN and LMM programs require no special treatment to be loaded this way but XMM programs have to be compiled using special command line options. Also, the companion load program **Generic_SIO_Loader** must be running on the target CPU – see below for more details on this.

Refer to the **Catalyst Reference Manual** for more details on compiling, installing and using Catalyst.

Generic_SIO_Loader

This is the companion loader program that must be running on a CPU for the Generic SD Loader program to be able to load programs to that CPU. The easiest way to make sure this program is running on the target CPU is to program it into the EEPROM of the CPU – but if the CPU has no EEPROM, this program can also be

downloaded from a PC. Or it can be downloaded from another CPU using the Generic SD Load program, by using the **CPU_Boot** programs (described below).

There will usually be a separate *Generic_SIO_Loader_n.spin* program (where n = 1, 2 or 3) for each CPU in a multi-CPU system that might need to be loaded from the Prop with the SD card

CPU_n_Boot

This is a self contained boot load program that can be used to load the companion loader onto the CPU. This program does not itself need to be loaded into the other CPU (that would be a chicken and egg problem!) instead, it is loaded and executed on the local CPU using the **Catalyst** SD Loader – when executed, this program reboots the other CPU and then uses the Propeller's built-in boot loading process to load the **Generic_SIO_Loader** program into that CPU – after that, the Catalyst program can be used again to load any arbitrary program into the remote CPU.

There will usually be a separate *CPU_n_Boot_m.spin* program (where n, m = 1, 2 or 3) for each CPU in a multi-CPU system that might need to be loaded from the Prop with the SD card (which would normally run Catalyst).

CPU_n_Reset

This is a utility program that can be loaded into the local CPU. When executed it simply resets the remote CPU – this is a useful alternative to power-cycling the whole multi-CPU system just because one CPU is not responding.

There will usually be a separate *CPU_n_Reset_m.spin* program (where n, m = 1, 2 or 3) for each CPU in a multi-CPU system that might need to be reset from the Prop with the SD card (which would normally run Catalyst).

Multi-CPU Examples

The best way to understand loading CPUs in a multi-Prop system is to try out a few cases – see the document **Getting Started with Catalina** for a tutorial containing some examples.

Parallelizer Support

Catalina now supports a pragma preprocessor which can be used to turn a serial C program into a parallel C program, without modifying the C source code. Instead, *pragmas* can be added to the source code to tell Catalina how to distribute the program across multiple cogs. The easiest way to illustrate this is with a simple example.

Below is a fairly trivial “Hello, world” type program:

```
void main() {
    int i;

    printf("Hello, world!\n\n");

    for (i = 1; i <= 10; i++) {
        printf("a simple test ");
    }

    printf("\n\nGoodbye, world!\n");

    while(1); // never terminate
}
```

And below is how the program might appear with the addition of a few pragmas (shown in green).

```
#pragma propeller worker(void)

void main() {
    int i;

    printf("Hello, world!\n\n");

    for (i = 1; i <= 10; i++) {

        #pragma propeller begin
        printf("a simple test ");
        #pragma propeller end

    }

    printf("\n\nGoodbye, world!\n");

    while(1); // never terminate
}
```

Note that we have not modified the original source code at all. But the effect of adding these pragmas is that, when the program is compiled with the **-Z** option to Catalina, all the instances of the **for** loop – i.e. the **printf** statements – will be executed *in parallel*, using all the available cogs on the Propeller.

The Catalina Parallelizer is fully described, with tutorial examples, in the document **Parallel Processing with Catalina**. Refer to that document for more details.

Customizing Catalina

Customized Platforms

Catalina allows new platforms to be supported very easily. Creating a new platform gives you the opportunity to specify the pin and clock configurations, and also the HMI options that the platform supports (e.g. whether it supports both TV and VGA output, or only one of these, or neither).

Each platform has a symbol reserved for it (e.g. **HYDRA**). One symbol (**CUSTOM**) is the default, and is intended to be suitable for nearly any Propeller with a 5 Mhz clock.

To modify the details of a **CUSTOM** platform, the following files in the Catalina standard target package (in the *target* sub-directory) need to be modified:

- Custom_DEF.inc* - specifies pin and clock configuration
- Custom_CFG.inc* - specifies plugin configuration options
- Custom_HMI.inc* - specifies the HMI supported option
- Custom_XMM.inc* - specifies the XMM API

To use any platform except the CUSTOM platform, you define the symbol on the command line. While not necessary, it does no harm to define the symbol **CUSTOM**. For example, the following two commands are equivalent:

```
catalina hello_world.c -lc
catalina hello_world.c -lc -C CUSTOM
```

If you want to add your own symbol (i.e. instead of **CUSTOM**) to support a new platforms (or a variations of a platform – e.g. to support a different clock speed, I/O configuration or XMM API) then the files *DEF.inc*, *CFG.inc*, *HMI.inc* and *XMM.inc* can be modified to include the new platform name.

For example, here is the relevant code from *HMI.inc*:

```
#ifndef HYDRA
#include "Hydra_HMI.inc"
#elseifdef HYBRID
#include "Hybrid_HMI.inc"
#elseifdef C3
#include "C3_HMI.inc"
#elseifdef TRIBLADEPROP
#include "TriBladeProp_HMI.inc"
#elseifdef RAMBLADE
#include "RamBlade_HMI.inc"
#elseifdef RAMBLADE3
#include "RamBlade3_HMI.inc"
#elseifdef DEMO
#include "Demo_HMI.inc"
#elseifdef DRACBLADE
#include "DracBlade_HMI.inc"
#elseifdef ASC
#include "ASC_HMI.inc"
#elseifdef PP
```

```
#include "PP_HMI.inc"
#ifdef QUICKSTART
#include "QuickStart_HMI.inc"
#endif
#ifdef ACTIVITY
#include "Activity_HMI.inc"
#endif
#ifdef FLIP
#include "Flip_HMI.inc"
#endif
#ifdef CUSTOM
#include "Custom_HMI.inc"
#else
' default is CUSTOM
#include "Custom_HMI.inc"
#endif
```

Simply add a new clause to the if statement in each of the four files (*XMM.inc* only if the platform has support for XMM). Your new platform will automatically be supported by all the targets in the target package.

The document **Getting Started with Catalina** has more details.

Customized Targets and Target Packages

Catalina also allows the creation of customized targets, and also entire target packages. Creating a customized target package for Catalina is fairly easy. To create a new target package, simply copy and rename a whole existing target directory, then make any changes you want.

New targets within a target package can be created by copying and modifying the files of an existing target. Note that new targets are normally required **only** if you need to define a new memory model, or perhaps need to support a very unusual plugin, driver or initialization code (for instance, the POD debugger is supported by defining a new target called **pod**). Normally, a new target is **not** required just to support a new propeller platform (as described in the previous section).

Each target is normally part of a target package, located in a specific directory under the Catalina base directory. The target is specified when binding or compiling by using the **-t** option to the Catalina Binder, or the Catalina Compiler. For example, to use a target called **my_target** the commands might be:

```
catbind my_file.s -t my_target [other_options]
```

or

```
catalina my_file.c -t mytarget [other_options]
```

To use targets in a different package, also use the **-Tpath** option to specify the path to the target *package*. For example:

```
catalina my_file.c -T mypackage -t mytarget [other_options]
```

Each Catalina target consists of several files:

```
lmm_<name>.spin    - and -
emm_<name>.spin    - and -
smm_<name>.spin    - and -
```


<code>xmm_<name>.spin</code>	These Spin files are the first file executed when the Catalina program is run. There is one for each of the memory models/load options currently supported by Catalina. These files load the program and any plugins required, and can be customized to load other cog programs if required. Then they invoke the required loader and/or kernel.
<code>catalina_<name>.s</code>	This LMM PASM file can be used to include target-specific PASM functions that need to be made available to the Catalina program. Note that any such functions must adopt the Catalina calling conventions.

Note that the default target and standard target package do not need to be specified when compiling or binding – they are used if no other target or package is specified.

Note that all targets get their hardware specific configuration data (such as the clock frequency and I/O pin definitions) from the file called *Catalina_Common.spin* in the appropriate target sub-directory. This file in turn includes various platform specific files. This means that there is often only *one* file that needs to be modified to add support for a new Propeller platform to all targets.

Catalina now comes with three target packages. These are described in the section called **Catalina Targets**. If you need to create a new customized target package for a specific application or a specific Propeller platform, the *embedded* target package would usually be the best choice to start from. It omits much of the complex and esoteric functionality, such as the many different types of HMI drivers, and the proxy drivers. It would generally be easier to take the embedded package and *add* a specific HMI driver, than take the standard target package (i.e. *target*) and *remove* all the unnecessary ones.

Using existing Parallax Drivers

If there is an existing Parallax driver for a particular device or platform, chances are that it can be used with Catalina without much trouble.

There are now two different ways of doing this, described in the sections below.

Use a Spin object unmodified

You can use a Spin object (such as one from the Parallax Object Exchange, or OBEX) as a Catalina plugin completely unmodified in many cases. If the driver does not require any interaction with C, then it is simply a matter of loading and starting it. If it requires interaction, it is normally quite simple to write a “wrapper” object (also in Spin) that is responsible for the interaction – e.g. communicating via the registry.

To use such objects, you compile them to an object format using any normal Spin compiler, then use the Catalina **spinc** utility to turn the objects into a form that can be loaded by Catalina at run-time. For examples of using the **spinc** utility, see the programs in the *demos/spinc* directory. That directory also contains a complete

example of replacing the normal Catalina HMI plugin with two Spin objects taken direct from the OBEX (a TV driver and a keyboard driver).

The main restriction on the Spin objects that can be used is that it should not overwrite any fixed areas of memory outside the areas defined by its VAR blocks. Some objects also write to their DAT blocks – this is fine provided it is only done during initialization.

The cost of using a Spin object as a plugin (rather than a PASM object, as described in the next section) is that Spin objects will generally exhibit slower performance and consume more hub RAM. However, if the driver is implemented largely, or entirely in Spin then this is the only available option.

Use only the PASM portion of the driver

Existing Spin drivers that are implemented mostly in PASM (usually with just a few Spin interface routines) can be turned into a Catalina PASM plugin with fairly minor code changes.

PASM plugins generally consume less resources, are faster, and are also easier to interact with from C than the original Spin/PASM objects.

Catalina itself uses PASM only versions of the standard Parallax drivers for many of its own plugins (e.g. the HMI plugins).

Normally, the PASM code does not need to be modified very much - the modifications required are mostly to the Spin code, and consists of:

- Removing the unused Spin methods (usually everything except the *Start* method) – these are then generally replaced with C equivalents;
- Replacing any use of VAR blocks with a configurable data block (allocated by Catalina) which is passed to the PASM code on initialization;
- Using the registry for any necessary interaction with Catalina.

The Catalina HMI, Floating point, and SD card plugins give examples of the types of modifications required.

A fully-worked example is the gamepad driver (new with release 3.0). This driver was derived from the Spin/PASM driver provided for the Hydra. The code modifications required to convert the original Hydra driver (i.e. *gamepad_drv_001.spin*) to a Catalina Plugin (i.e. *Catalina_Gamepad.spin*) have been highlighted in the modified version to make it easy to understand.

In this case, the standard C library also contains a couple of functions to simplify the use of the driver (these are defined in *catalina_gamepad.h*) but these are completely trivial functions that just “wrap” the standard C function used to access the registry (e.g. ***_short_plugin_request***).

Loading and starting such additional plugins has been made much simpler since all target files in the Catalina standard target package now include a Spin file called *Extras.spin* – this means that you only need to modify a single file to add a new

plugin to all targets in the package. Examine *Extras.spin* to see how the gamepad plugin was added, and also how it has been associated with the **GAMEPAD** command line symbol.

The game pad plugin can be demonstrated by the program *test_gamepad.c* (note – this program has only been tested on the Hydra!):

```
catalina -lci test_gamepad.c -C GAMEPAD
```

Building Catalina

There are detailed instructions on building Catalina from source for Windows and Linux in the document *BUILD.TXT* in the Catalina directory.

The Catalina Windows binaries were compiled on a 64-bit version of Windows 10 but should work Windows 8 or later.

The Catalina Linux binaries were compiled on a 64-bit version of Ubuntu Linux 18.04, but may work on other Ubuntu distributions.

Some previous releases of Catalina have been built on Apple OSX (Darwin) but the current release will need quite a lot of work to do so.

Note that on Windows it is unlikely you will need to rebuild Catalina, since it does not need to be rebuilt to compile programs, or add new platforms, new plugins or new libraries.

On Linux it is more likely, since Catalina may need to be rebuilt to suit other Linux distributions.

Catalina Technical Notes

This section contains technical notes about various aspects of Catalina.

A Note about Binding and Library Management

Catalina uses **lcc** as its C compiler, providing a custom code generator specific to the Parallax Propeller. Catalina also replaces the normal *linker* that **lcc** expects with a *binder*. A binder does a similar job to a linker, but works at the source code level instead of at the object code level – i.e. instead of the usual *compile-assemble-link* sequence, Catalina uses a *compile-bind-assemble* sequence.

The Catalina Binder is called **catbind**.

To understand the relationship between **catalina**, **lcc** and **catbind**, consider the following steps involved in generating a binary file from a C source file using Catalina:

- The **catalina** program processes the command line and parses the command line options and the environment variables to determine a set of options to be passed to **lcc**. It then invokes **lcc**.
- **lcc** preprocesses the C source file (which has a **.c** extension) to expand all *macros* and *include* files, parses the resulting C source file for validity, and produces a syntax tree of the C program.
- **lcc** traverses the syntax tree, invoking the Catalina code generator on each node as required to produce LMM PASM statements that are the logical equivalent of the original C program – the result is written to an LMM PASM source file (with a **.s** extension).
- Because Catalina binds at the source level, but **lcc** expects to invoke a linker that binds at an object level, the Catalina version of **lcc** simply renames the LMM PASM file (**.s** extension) to appear to be an “object” file (**.obj** extension under Windows).
- **lcc** invokes **catbind** on the “object” file (actually an LMM PASM source file) to combine this source file with other source files from various libraries – the binder recursively resolves all the source symbols by including library files until all symbols in all the included files have been resolved – the result is then output as a single SPIN/PASM file (with a **.spin** extension).
- For LMM programs, the **catbind** program in turn then invokes the **spinnaker** SPIN compiler on the nominated target SPIN file (not directly on the binder output). LMM target SPIN files are normal Propeller SPIN program that refers out to the following objects in other files²⁰:
 - o The SPIN file created by the binder (i.e. the user program);
 - o The Catalina Kernel;

²⁰ EMM and XMM targets work slightly differently. Refer to the sections on EMM Support and XMM Support.

- o Any plugins required by the target;
- **spinnaker** assembles the target SPIN/PASM file, and typically produces either a binary file (with a **.binary** extension) or an eeprom file (with a **.eeprom** extension). A listing file can also be produced (with a **.lst** or **.list** extension).

Part of the job of **catbind** is to resolve any references in the original C program to functions provided by external libraries. Catalina libraries are simply collections of PASM source files produced by using **lcc** – but without binding or compiling the resulting output (this is done by using the **-S** option to **lcc**). The standard C89 libraries provided (e.g. **libc**, **libm**) are generated using **lcc** in this way.

To enable libraries to be efficiently searched when the binder needs to resolve symbols, each library must have an index. **catbind** is itself used to produce these indexes. This means that user libraries must be created using **catalina**, and then indexed using **catbind**.

The **catbind** program can also be used to provide diagnostic help for determining how a symbol has been resolved, or in determining where an unresolved symbol is referenced.

A Note about the Catalina Libraries

Catalina provides a complete set of ANSI compliant C89 libraries, with some C99 additions. There are several different versions of each of the libraries provided:

- libc** the standard C library. This version of the library supports only *stdin*, *stdout* and *stderr* – it does not support full file system access, and is appropriate for platforms with no SD Card file system (or for programs that do not need to use the file system).
- libci** the standard C library without floating point support in *stdio* (e.g. in routines such as *printf* and *scanf*). Like **libc**, this version of the library supports only *stdin*, *stdout* and *stderr* – it does not support full file system access, and is appropriate for platforms with no SD Card file system (or for programs that do not need to use the file system) and which do not need to do input or output on floating point numbers. Note that **sprintf** and **vsprintf** are not supported by **libci**. If these are required, use one of the other libraries instead. This library is significantly smaller than **libc**.
- libcx** the standard C library with extended file system support. This version of the library is appropriate for programs which need to use an SD Card file system. This library is significantly larger than **libc**, and should only be used where SD card file system access is required.
- libcix** the standard C library with extended file system support, but without floating point support in *stdio* (e.g. in routines such as *printf* and *scanf*). This version of the library is appropriate for programs which need to use an SD Card file system, but which do not need to do input or output on floating point numbers. This library is significantly smaller than **libcx**.
- libm** the standard maths library, emulated in software. This version of the maths library does not require any extra cogs, but is larger and slower than the other versions.
- libma** the standard maths library, with some functions emulated on a separate cog. This library is smaller and faster than **libm**, but requires a free cog (this is transparent to the user of the library - the management of the cog is handled automatically by Catalina).
- libmb** the standard maths library, with more functions emulated on two separate cogs. This library is smaller and faster than **libma**, but requires two free cogs (this is transparent to the user of the library – the management of the cogs is handled automatically by Catalina).

Note that as well as all the standard C functions, **libc**, **libci**, **libcix** and **libcx** also provide the standard Catalina HMI functions described in the **HMI Support** section.

Note that there are two complete sets of the above libraries provided with Catalina – one is contained within the **lib** sub-directory, and the other is contained in the **large_lib** sub-directory. The reason for this is that programs built using the LARGE

addressing model (i.e. with the **-x5** command line option) also require the library code to be generated with this option. The large address model actually uses a completely different code generator to the tiny and small memory models. Catalina automatically selects the correct set of libraries based on the command line options (e.g. the **-x5** option). Note that this means ***you cannot combine user-created libraries and programs compiled using different addressing modes, except as follows:***

- programs built with **-x3** can use libraries built with **-x5**
- programs built with **-x4** can use libraries built with **-x2** or **-x0**

A Note about C Program Startup & Memory Management

Catalina memory management is reasonably simple. When a Catalina program is loaded into Propeller RAM it consists of a collection of SPIN or PASM objects, the same as any other Propeller program. The objects in the Catalina program typically end up being loaded in the following order:

- The compiled Catalina C program (LMM or CMM PASM)
- The Catalina Kernel (SPIN/PASM)
- Any plugins specified by the Target program (SPIN/PASM)
- The specified Target program (SPIN/PASM)
- VAR space for all SPIN objects

After this comes unallocated space, up to the top of RAM. On the Propeller 1 this is 32 kb or \$8000).

On startup, it is the SPIN/PASM *target* program that executes first. This program is responsible for loading the Catalina Kernel into cog RAM and starting it. It then loads each of the required plugins into cog RAM and starts each one. Then it starts the Kernel. Finally, it terminates its own cog, shutting down the SPIN interpreter. From this point on only the Kernel and the plugins are executing. Theoretically, either the Catalina program, or one of the plugins could then (but currently do not) re-use the terminated cog for other purposes.

Catalina uses the upper end of RAM as data space for plugins. Any SPIN objects that represent plugins may use VAR space while they are being initialized by the target program, but once the Kernel has been started a plugin must no longer use any VAR space. To make this possible, the Target program allocates each plugin a data block at the upper end of RAM, and passes the plugin its data block address as part of the startup process. This is the main reason that a standard Parallax driver typically needs to be modified for use with Catalina.

The stack used by a Catalina program starts just below the plugin data and grows downwards in memory. The heap used by a Catalina program starts just above the Catalina program itself and grows upwards in memory. This means that the heap and the stack may both include space that was originally occupied by the Kernel, a plugin or a driver. This is fine because once they are loaded and executing, each plugin only uses the data block assigned to it by Catalina – this means neither the Kernel nor the plugins require this space any longer, and it can be reallocated for use by the Catalina program.

This also explains why trying to save RAM space by eliminating unused drivers is often not required – such space is reclaimed automatically to be used as Catalina heap and stack space anyway. The only time it can be useful to eliminate unused drivers is when leaving them makes the final program too large to load in the first place, or too large to use with the POD debugger.

There are a few Catalina-specific additions to the standard C memory management functions provided by the C library (i.e. **malloc**, **calloc**, **realloc**, **free**):

1. The configurable constant `MIN_SPLIT` has been introduced (defined in *impl.h* in the standard library sources). This is the smallest amount that must be left before a memory block will be split in two if it is used to allocate (or reallocate) a memory block smaller than its actual size – otherwise the block is left intact. This can significantly reduce memory fragmentation, which can lead to no memory blocks of sufficient size being available to satisfy a memory allocation request, even though there is plenty of memory free. `MIN_SPLIT` must be a power of 2. By default it is set to 64 (bytes).

Note that the memory management constant `GRABSIZE` (the minimum size that **malloc** is configured to grab whenever it needs more RAM space) has also been set to `MIN_SPLIT`. In previous versions of Catalina it was set to 512, but this is too large for programs (like Lua) that do extensive allocation of small memory blocks. The previous value can be restored if required (edit *impl.h*) but the Catalina libraries will have to be recompiled.

2. The internal memory defragmenter has been made externally visible as **malloc_defragment()** - this allows the defragmenter to be manually invoked periodically to recover from extreme memory fragmentation should it occur. Doing so is typically only required by programs that allocate, free and/or reallocate many small randomly sized chunks of memory – programs that mostly allocates fixed block sizes (e.g. buffers of a standard size) will not be affected and don't need to manually run the defragmenter. The problem occurs mostly with multi-threaded programs because the stack requirements of such programs tends to be much larger, but there is no easy way to tell when heap memory is getting low, which is the only time the original memory management code called the defragmenter.
3. The standard C memory management functions (**malloc**, **realloc**, **calloc**, **free**) and the non-standard system break function (**sbrk**) are thread-safe – this is not required by the ANSI C standard, which has no concept of threads. Since this incurs a small performance penalty, this is done by default only for programs that also use multi-threading, where it is most likely to be an issue – but in all cases you can decide to have these functions either made thread-safe or not. Even in a multi-threaded program, if only one thread does memory allocation then there is no need for a lock and you can decide not to make these functions thread-safe if you find it affects your program performance.

If your program already uses the thread library, you will probably not need to do anything to make your memory management thread-safe. Such programs should already call the **_thread_set_lock()** function, which will now also allocate a memory lock (unless one has already been allocated).

To specify a memory lock should be used in other situations (such as in multi-cog or multi-modal programs where more than one program might use

dynamic memory allocation), or to manually specify the lock to use, first allocate a lock using the normal **_locknew()** function, and then use the function **_memory_set_lock()** which tells the memory management functions to use the specified lock. Note that the lock only has to be set by one program even if there are programs running on multiple cogs and/or in multiple memory models – the lock itself is stored in a global variable and will be used by all the Catalina programs currently executing.

You can just retrieve the memory management lock that is currently in use without affecting anything by calling the function **_memory_get_lock()**. It will return -1 if no memory management lock is in use.

Note that to use a specific lock in a multi-threaded program, you must call **_memory_set_lock()** *before* calling **_thread_set_lock()**.

You can stop using the current memory lock by calling **_memory_set_lock()** with a value of -1, but note that this does not release the lock – to do that you can either use the value returned from this function (which will be the lock that was in use or -1 if none was in use) in a call to the **_lockret()** function.

If you use **_sbrk()** you can continue to do so, but it remains potentially thread unsafe. But there is now also a thread-safe version called **sbrk()** (i.e. without the underscore) so if you use multi-threading you may want to modify your programs to use that function instead.

These non-standard memory management functions are defined in the include file *catalina_cog.h*, and the various thread functions are defined in *catalina_thread.h*.

There is a demo program that tests the thread-safe **malloc** in the *demos\multithread* folder – called *test_thread_malloc.c*

A Note about POD and EMM/XMM

There are currently no EMM or XMM ‘debug’ targets. **POD** cannot be used to debug EMM or XMM programs. **POD** is partially implemented in SPIN, and because the EMM and XMM Loaders rearrange the Propeller memory in order to make the entire 32K of onboard RAM available to Catalina programs, SPIN programs can no longer be executed in conjunction with Catalina EMM or XMM programs (although SPIN is still used to load and initialize Catalina plugins before the EMM or XMM Loader takes over).

POD targets for EMM and XMM may be included in a subsequent of Catalina – now that the LARGE memory model allows C program data segments to be moved out of HUB RAM, it may in future be possible to have some part of the HUB RAM reserved exclusively for use by SPIN programs.

In the meantime, for source level debugging, **BlackBox** can debug LMM, EMM or XMM programs. Alternatively, programs can initially be developed and debugged as LMM programs, with only the final version compiled as an EMM or XMM program.

A Note about Catalina Code Sizes

Given the limited amount of RAM on the Propeller (32 kb) it is easy to generate programs where the binary may initially seem to end up too large to be useful.

Having the ability to run Catalina programs from external memory (XMM) is one option, but not all platforms support XMM RAM. On platforms with only the in-built 32 kb of Hub RAM Catalina offers many alternatives to reduce the final program size.

First, consider the program below – it is the traditional C “Hello, world” program:

```
#include <stdio.h>
void main() {
    printf("Hello, world (from Catalina!)\n");
}
```

To compile this program (a version of it is actually included in the *Catalina\demos* folder), we might initially use a simple command such as:

```
catalina hello_world.c -lc
```

This compiles the program, links it with the **libc** version of C89 library and uses the default target and drivers, and the LMM kernel. Here are the statistics actually produced by the above command:

```
code = 17452 bytes
cnst = 144 bytes
init = 216 bytes
data = 1068 bytes
file = 24016 bytes
```

At first sight, it appears that this trivial program consumes nearly **20 kb** of our precious 32 kb of inbuilt Hub RAM! Don't worry - this can be substantially reduced in various simple ways.

For example, by using the **libci** library (in place of **libc**) for programs that do not require support for i/o of floating point numbers (they can still use floating point internally – they just can't scan or print them) the program size immediately reduces from **20 kb** to around **12 kb**. To see this, compile the same program using this command instead:

```
catalina hello_world.c -lci
```

Here are the statistics:

```
code = 6232 bytes
cnst = 103 bytes
init = 208 bytes
data = 772 bytes
file = 12452 bytes
```

The exact values may vary depending on your version of Catalina, or if you have set up a different default platform or target, or are using the **CATALINA_DEFINE** environment variable.

The big difference in code size is due to how much code is required in the standard C library to perform I/O of floating point numbers.

If you don't need the full capabilities of the Standard C I/O libraries, Catalina also provides smaller alternatives. In this case, we can use the “tiny” library, which provides smaller (but more limited) versions of the standard C I/O functions. We specify those in addition to the standard library:

```
catalina hello_world.c -lci -ltiny
```

Here are the statistics:

```
code = 2956 bytes
cnst = 15 bytes
init = 224 bytes
data = 772 bytes
file = 9104 bytes
```

For this program, several other optimizations are also possible, and can be selected on the command line. For example, try the following command:

```
catalina hello_world.c -lci -ltiny -C NO_FLOAT -C NO_ARGS -C NO_EXIT
```

Here are the statistics:

```
code = 2904 bytes
cnst = 15 bytes
init = 220 bytes
data = 772 bytes
file = 8120 bytes
```

Our program size is now around **8kb**. But that's not the end of the story. Notice that the sum of the segments (**code**, **cnst**, **init** & **data**) is actually under **4 kb**? The other **4 kb** of the final file size is taken up by plugins and the kernel itself. But we don't need to waste even this space - Catalina provides EMM (EEPROM) and SMM (SDCARD) loaders which can make this space available as code space. So in reality, the program requires only **4 kb** of Hub RAM *even though it includes a substantial portion of the stdio library* (which of course only ever needs to be included once). This leaves us around **28 kb** of Hub RAM for more C code (or for other purposes). And that code does not need to include the stdio library code again!

To make the significance of this last point more evident, consider the following program, very similar to the traditional C “hello, world” program:

```
#include <catalina_hmi.h>
void main () {
    t_string(1, "Hello, world (from Catalina!)\n");
}
```

Let's compile this program with our next command (this program is also in the Catalina\demos folder, called *hello_world_3.c*):

```
catalina hello_world_3.c -lci -C NO_FLOAT -C NO_ARGS -C NO_EXIT
```

Here are the statistics:

```
code = 204 bytes
cnst = 15 bytes
init = 4 bytes
data = 4 bytes
file = 4436 bytes
```

Excluding the “one off” Catalina runtime overheads (i.e. the Catalina kernel, the HMI plugin and various support functions) this program actually compiles to only around 250 bytes, even though when compiled and bound with a suitable target, it occupies about **4.5 kb** of RAM - this is because (in this case) Catalina must also include at least the following:

- the Catalina Kernel (~2 kb);
- a Catalina HMI plugin and drivers (~2.5 kb);

But if we were to load this program using an SMM or EMM two-phase loader, it requires *only the 250 bytes* of Hub RAM – leaving over **31 kb** for more code!

The difference between the two “hello world” programs is that the first version uses the **printf** function from the C89 library, whereas the second uses a **t_string** function which is built into the Catalina HMI plugin. This small change means that Catalina does not need to load a significant part of the C library (i.e. **printf** and its associated support functions). In reality, this small difference between the two programs amounts to over a thousand lines of library C code. But many programs do not require the full functionality provided by **libc**, and some do not need to load it at all.

EMM targets can be used on any Propeller with a 64 kb EEPROM attached, and SMM targets can be used on any Propeller with an SD Card attached. While these techniques do not reduce the program size, they do mean that more of the available 32 kb of RAM can actually be used by the C program.

As if this is not enough, there is also the Catalina Compact Memory Model (CMM). This dramatically reduces the size of any Propeller programs – typically by over 50% - but at a cost in reduced speed. The resulting programs will still execute faster than Spin, but slower than the equivalent LMM program.

Let’s compile the same program in COMPACT mode:

```
catalina hello_world_3.c -lci -C NO_FLOAT -C NO_ARGS -C NO_EXIT -C COMPACT
```

Here are the statistics:

```
code = 104 bytes
cnst = 15 bytes
init = 4 bytes
data = 4 bytes
file = 4412 bytes
```

Finally, there is also the Catalina Code Optimizer that can reduce code sizes even further – typically 15 – 20% for LMM programs (slightly less for CMM programs). The Catalina Optimizer works on any size program. For example, here are the statistics when both CMM and the Optimizer are used on the above program:

```
catalina hello_world_3.c -lci -C NO_FLOAT -C NO_ARGS -C NO_EXIT -C COMPACT -O5
```

Here are the statistics:

```
code = 88 bytes
cnst = 16 bytes
init = 0 bytes
data = 4 bytes
```

file = 4392 bytes

Our final Catalina C program, which implements the same functionality as our original program, may now use as little as 108 bytes of Hub RAM to execute. We can use a combination of these code size reduction techniques with nearly all C programs.

A Note about Catalina symbols vs C symbols

Both Catalina and **lcc** make use of symbols. Catalina uses symbols to pass configuration options to the various targets compiled by the **spinnaker** Spin compilers (remember, Catalina targets are really just SPIN programs). **lcc** uses symbols to pass configuration options to the C program it is compiling.

Is there a relationship between these symbols? Obviously you can access Catalina symbols from within the targets (that's what they're intended for!) but can you also access such symbols from within a C program? The answer is 'yes' – but with a few minor complications.

The Catalina front-end accepts symbols defined on the command line using the **-C** command line option, or specified in the **CATALINA_DEFINE** environment variable. When a symbol is defined two things happen:

1. The symbol is passed to the Catalina Binder, and is then passed on to Spinnaker to conditionally compile based on the symbol defined on the Catalina command line.
2. The symbol is passed to **lcc**, but with the prefix **__CATALINA_** added. This is done to avoid name collision with names commonly used in C programs. This symbol will be available to C programs like any other.

So when compiling a program using a command like this:

```
catalina hello_world.c -lc -C HYDRA -C VGA
```

Then the following happens:

1. The following symbols will be passed through **lcc** to the Catalina Binder, and can be used in targets to conditionally include or exclude sections of Spin or PASM code:

```
HYDRA
VGA
```

2. The following symbols will be passed to **lcc**, and can be used in C programs like any other C symbol:

```
__CATALINA_HYDRA
__CATALINA_VGA
```

Note that to define symbols from within either Spinnaker or C and then have them available to programs written in the other language is *not* currently supported.

It is possible to pass symbols directly to C programs (i.e. without Catalina adding in the **__CATALINA_** prefix) by using the appropriate Catalina command line option (e.g. **-DXXXX**), or using the **CATALINA_LCCOPT** environment variable (e.g. **set CATALINA_LCCOPT= -DXXXX**). This environment variable is used to pass options directly to **lcc**. However, it is important to note the difference between this environment variable and the **CATALINA_DEFINE** environment variable – i.e.:

Symbols defined using the **CATALINA_LCCOPT** environment variable **must** be preceded by **-C** – this is because this environment variable can contain

any **lcc** option, not just symbol definitions. Within the C program, these symbols will appear “as is” (i.e. without any Catalina prefix added).

Symbols defined in the **CATALINA_DEFINE** environment variable ***must not*** be preceded by **-c** because this environment variable is processed by Catalina and can contain only symbol definitions. Within Spinnaker programs these symbols will appear “as is” but within C programs these symbols will appear with the standard Catalina prefix added.

Finally, note that Catalina always defines the symbol **__CATALINA__** – this symbol can therefore always be used within C programs to determine if they are being compiled with Catalina.

A Note about the Catalina Loader Protocol

Catalina uses both the normal Parallax load protocol (e.g. to load an LMM program from a PC) as well as its own internal protocol (e.g. to load an XMM program from a PC, or to load programs between CPUs).

The loader protocol is a single-master, multi-slave protocol, where the master can be either a PC, or one of the on-board CPUs (usually the one with direct access to the SD card). All the slave CPUs in the system are expected to monitor the serial line and read all packets – but ignore them until they see their specific sync signal. They respond to the master using the same sync signal.

Note that CPU numbers always start from 1 (e.g. 1, 2 & 3 on a TriBladeProp) – there is no CPU 0.

All file loads start with the “sync” signal. The sync signal is always two bytes - **\$FF \$nn**, where **\$nn** is the number of the CPU for which the data is intended (if requesting) or the CPU from which it is sent (if responding).

For example, to alert CPU #2 to receive a file, the sync signal **\$FF \$02** is sent. During the transfer, whenever CPU #2 needs to respond, it sends **\$FF \$02** back again (before each response).

Byte stuffing is used to prevent sync signals being interpreted accidentally (e.g. within a stream of binary data). This means that other than within the sync signal itself, any transmission of a single byte **\$FF** is "stuffed" to send two bytes - **\$FF \$00**. Any such sequence seen by the receiver must be "unstuffed" back to a single **\$FF** (note that **\$FF \$00** can never represent a sync signal, as there is no CPU 0).

To send a file, a “sync” is first sent to the destination CPU. Then the file is sent in packets. Each packet has the following format:

- <address>** 4 bytes, with the top byte containing the CPU number for which the packet is intended, and the lower 3 bytes containing 24 bits of address. The address **\$FFFFFFE** reserved as an EOT marker
- <size>** 4 bytes specifying the size of the following binary data
- <data>** up to <size> bytes of data (usually one sector)

Note that each packet should NOT begin with another sync signal – this would initiate a new transfer.

The CPU receiving the packets responds to each packet with:

- <sync>** a sync signal containing its own CPU #
- <LRC>** single byte LRC of the <size> data bytes it just received.

If the sender fails to receive this response to each packet, or the LRC does not match, or a timeout expires, then the packet is retransmitted. Otherwise the next packet is transmitted. To complete the transmission, a special "empty" data packet is sent - i.e.:

\$00FFFFFFE the special address marker that means EOT

\$00000000 zero bytes follow

At any time, a transmission can be aborted simply by not sending any more packets. All receivers start the process all over again whenever they see another sync signal – even if it is in the middle of a packet transmission.

The master may send an initial “dummy” packet, specifying address \$000000, which should be processed by the slave like any other packet. Since all Catalina payload loads will include a real packet for this address, it does not generally matter if this dummy packet is processed by the slave or not. The dummy packet is used by the master to detect whether a slave is present, since receiving the response sync signal alone (which might simply be echoed by another serial program) is not sufficient.

Payload uses a packet size of 512 bytes, which is typically one disk sector for all packets, including the dummy packet. However, other packet sizes may be used and should be supported by a slave protocol implementation.

Catalina’s payload loader contains an implementation of the master side of this loader protocol, and there is an example of implementing the slave side of this protocol in the file *load.c* in the *demoss/loader* directory.

Catalina Development

Reporting Bugs

Please report all Catalina bugs to ross@thevastydeep.com.

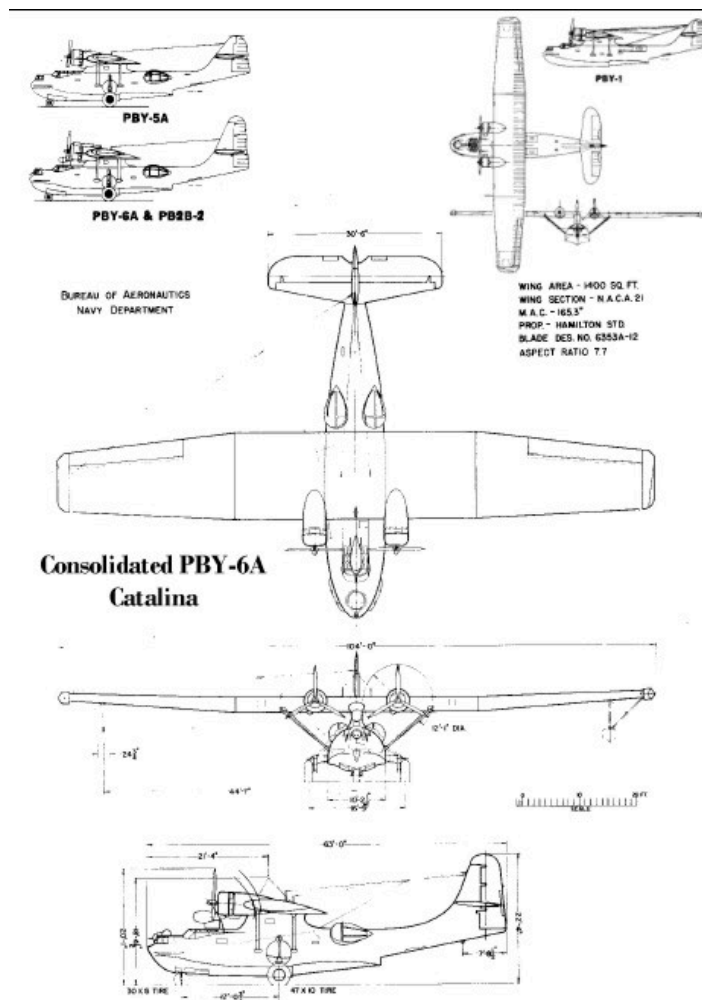
Where possible, please include a *brief* source code example that demonstrates the problem.

If you want to help develop Catalina

Anyone who has ideas or wants to assist in the development of Catalina should contact Ross Higson at ross@thevastydeep.com

Okay, but why is it called “Catalina”?

Why? Because it’s a big, slow, “C”-going contraption ... powered by Propellers!



See http://en.wikipedia.org/wiki/PBY_Catalina

Acknowledgments

Parallax – for creating the Propeller chip in the first place

Chris Fraser and **David Hanson** – for **Icc**

André LaMothe – for the Hydra platform, and also the C3 platform

Cluso99 – for the TriBladeProp and RamBlade platforms, and the Propeller 2 SD card functions.

Coley – for the Hybrid Propeller board

Dr_Acula – for the DracBlade platform

Bill Henning – for his original work on the Large Memory Model for the Propeller, and also for the Morpheus platform.

Mike Green – for the keyboard driver that supports both Hydra and Demo boards, and also for his basic I2C driver

Cam Thompson – for the Float32 libraries for the Propeller

Kaio – for the POD debugger

Insonix – for Prop Terminal

Brad Campbell – for bstc

Baggers – for the high resolution TV text driver

Michael Park – for Homespun

Lewin Edwards – for the DOSFS SD Card file system

Radical Eye Software – for the fsrw SD Card file system

Eric Moyer – for the modified firmware for the Hydra Xtreme HX512 SRAM card

Bob Anderson – for his work on the BlackCat debugger

Kye – for the FATEngine file system

Various contributors – for the C89 C library (see the source files)

Pullmoll – for the improved signed integer division routine

Hanno Sander – for his work on Catalina support in ViewPort

Steve Densen – for the original spinc utility, and his work on the caching SPI driver

David Betz – for his work on the caching SPI driver

Microcontrolled – for the Catalina logo

Rayman – for the FlashPoint XMM boards (SuperQuad, RamPage and RamPage 2)

Ted Stefanik – for procedures to manipulate the Propeller special registers, and for the libtiny library.

Nick Sabalausky - for the 22KHz, 16-bit, 6 Channels Sound Driver

Roy Eltham – for openspin, the open source version of the Parallax Spin compiler, from which spinnaker is derived.

Dave Hein – for the Propeller 2 p2asm assembler.

Ozpropdev – for the Propeller 2 Flash Loader.

Michael Sommer (Msrobots) – for the Propeller 2 2-port Full Duplex Serial Driver.

Catalina Internals

This section contains technical descriptions about various Catalina internals.

A normal user of Catalina who just wants to compile C programs does not need to know anything contained in these sections – they are provided for users who may want to know more about how Catalina works, or who may want to modify Catalina.

A Description of the LMM and XMM Kernels

The Propeller chip is a wonderful thing – eight 32 bit RISC processors sharing 64 kb of RAM, with all eight processors also sharing 32 general purpose I/O pins! But a significant limitation of the design is that each of the processors (or *cogs*, to use the Parallax terminology) only has direct, full-time access to 2048 bytes of *cog* RAM for the direct execution of Propeller Assembly (PASM) programs. The remaining 32kb of RAM (referred to as *hub* RAM) is shared amongst all the cogs on a round-robin basis, with each cog only being able to access this RAM 1/8th of the time. Also, the RAM dedicated to each cog is organized as 512 longs – of which only 496 can be used to hold instructions or general purpose registers.

To execute programs larger than 496 instructions, one (or more) of the cogs is typically allocated to running a built-in SPIN language interpreter, which executes byte-coded SPIN programs out of the shared 32 kb of hub RAM.

However, while a SPIN program can access the full 32 kb of hub RAM, SPIN is an interpreted language, and it is literally *dozens* of times slower to execute an instruction in SPIN than it is to execute the equivalent instruction in PASM.

The **Large Memory Model (LMM)** mode for the Propeller chip was originally proposed by Bill Henning as a means of allowing PASM programs to be larger than 496 instructions. Bill realized that a cog could be used to *first* fetch, and *then* execute PASM instructions *from hub RAM* using a few simple cog-based instructions –allowing arbitrary sized PASM programs to be executed. Essentially, LMM mode uses a cog to *simulate* a cog – and even though the simulated cog runs (at best) only ¼ the speed of a real cog, it has access to the full 32 kb of hub RAM for the storage of PASM instructions.

LMM mode is essential in enabling traditional high-level compiled languages to be run on the Propeller, since there are few compilers that can compile useful high-level language programs into only 496 instructions. Also, even though hub-based LMM programs are at least four times slower to execute than cog-based native programs (due to the fact that each instruction can only be fetched for execution while the cog has access to hub RAM), LMM programs are typically still many times faster than SPIN.

The basic code to implement LMM consists of a loop of only 4 PASM instructions - for more details see Bill Henning's original thread on the Parallax forums at <http://forums.parallax.com/forums/default.aspx?f=25&p=1&m=154421>. The program that executes these four instructions (in a simple loop) is referred to as an LMM *kernel*. Since it would be very wasteful to use an entire cog to execute only 4 instructions, most kernels use the remainder of the cog RAM to implement useful

primitive operations designed to simplify whatever job the LMM kernel is designed for. For example, most modern procedural computer languages make extensive use of both a call stack and a stack frame – so it is useful for an LMM kernel intended to support high level languages to implement both call/return and stack frame manipulation primitives – this not only simplifies the job of the language compiler, it also reduces the code size and improves the overall execution speed of the resulting program.

This is exactly what both the Catalina LMM Kernel does.

The Catalina LMM kernel can execute PASM programs up to 8192 instructions long – i.e. the number of instructions that can fit into the 32 kb of hub RAM built into the Propeller. While this is sufficient to enable many useful programs to be written in a high-level compiled language, it is still somewhat limiting. So the next step was to use some of the I/O pins on the Propeller to implement a general purpose external memory bus. When supported by appropriate external hardware, this allows PASM instructions to be stored ‘off-chip’, yet still be fetched fast enough to be useful. This is the so-called **eXternal Memory Model (XMM)**.

There are many different varieties of XMM, depending on the type of external hardware, how many I/O pins are dedicated to memory access, how the external memory must be addressed, and whether this access is ‘latched’ or not. Typical of XMM implementations is the Hydra Xtreme card – this card provides 512 kb of external SRAM, and uses 12 of the Propeller’s I/O pins - 8 pins are used to provide an 8 bit parallel bus, with 4 additional control lines necessary to control the access of data via this bus. The Xtreme was originally developed specifically for the Hydra games console, but is also now supported on several other Propeller platforms.

For the kernel, accessing external memory via XMM is even slower than accessing hub memory via LMM. For example, in the case of the Hydra Xtreme (with its 8 bit bus) it takes 4 separate byte-sized accesses to fetch each long PASM instruction for execution. But even so, executing PASM instructions from XMM can still be done fast enough to be useful. The architecture of the Propeller is such that anything that really needs to be done fast – such as a device driver – can have a cog completely dedicated to the job which runs a program direct from cog RAM. For most programs, the fact that the top-level program logic is executed more slowly – i.e. as LMM executing from hub either RAM or XMM RAM – is not usually an issue.

So the main difference between the LMM and XMM kernels is that the LMM Kernel is designed to execute LMM programs stored in hub RAM, while the XMM Kernel is designed to execute LMM programs stored in XMM RAM. The set of primitives supported by both Kernels is identical, although the means they use to implement these primitives sometimes differs.

The remainder of this document describes the XMM and LMM Kernels used to execute Catalina programs, by describing the common virtual machine registers, and the LMM primitives that both kernels implement.²¹

²¹ In this documentation, you may also see references to the **EEPROM Memory Model** (or *EMM*) mode. However, in EMM mode the Kernel used is still the LMM Kernel. The difference between

Note that this document describes the Catalina *beta* LMM and XMM kernels. These kernels (like the rest of Catalina) are still in beta release, and may change before the first ‘official’ release.

A Description of the Catalina Virtual Machine

The Catalina LMM and XMM Kernels both implement a 32 bit ‘virtual cog’, with various general purpose and dedicated registers which (unsurprisingly) is otherwise very similar to the 32 bit processor implemented by each physical cog.

This virtual cog supports a subset of existing PASM instructions, but this is extended by adding various LMM primitives to those instructions intended specifically to support Catalina C programs.

Registers

- r0 - r23** The kernel supports 24 **General Purpose Registers**. Each register is 32 bits, and each can hold a signed or unsigned integer, or a floating point value. Some of the kernel primitive operations make specific use of registers **r0** and **r1** to accept or return arguments (e.g. **MULT**, **DIVS**, **DIVU**). Additional conventions are imposed by the Catalina C compiler – e.g. the compiler always uses some of these registers as integer registers, and others as floating point registers. The compiler also uses **r2 .. r5** to pass the first 4 arguments to functions instead of passing them on the stack - this can make function calls much more efficient. However, these compiler conventions have nothing to do with the kernel itself, and may vary between compilers, or even between different versions of the same compiler.
- PC** The **Program Counter** holds a 32 bit pointer to the next instruction to be executed. Note that this is a byte address in either hub RAM (for the LMM kernel) or external RAM (for the XMM kernel). It is not a long address in cog RAM as it would be for a cog executing normal PASM. This means the PC must be incremented by 4 after each instruction, not by 1.
- SP** The **Stack Pointer** holds a 32 bit pointer to the top of the stack. The stack holds long values, and the SP points to the long that is currently on the top of the stack. The stack is implemented in hub RAM for both LMM and XMM programs. It is initialized on startup to point to the upper RAM area just below the configuration data blocks of various kernel plugins, and grows downwards in hub memory.

LMM mode and EMM mode is that in LMM mode both the cog initialization code and the program to be executed must be stored within the first 32 kb of external EEPROM, whereas in EEM mode only the cog initialization code is stored from the first 32 kb of the external EEPROM – the program to be executed is loaded from the *second* 32 kb - this allows EMM programs to be larger than LMM programs – although when executed, both execute from the Propeller’s hub RAM.

- FP** The **Frame Pointer** is a 32 bit pointer to the current execution frame, which is held on the stack. For a discussion of stack pointers and frame pointers, and the relationship and difference between them, see http://en.wikipedia.org/wiki/Call_stack. For specific details on the calling conventions used by Catalina, see the section on Catalina Calling Conventions below.
- RI** The **Intermediate Register** is a 32 bit register specifically used to pass or return a single 32 bit value to or from various LMM primitives. It is also used internally by many of the LMM primitives as a temporary register when calculating the absolute address of a **Relative Index** value, so programs should not expect **RI** to be preserved by any primitive.
- BC** The **Byte Count** register is a 32 bit register used internally by various primitives that move or copy structures – it represents the size of the structure (in bytes) to be moved or copied. It is also used to pass the size (in bytes) required by a function when creating a new frame, and to return the SP associated with the caller once the new stack frame has been established. However, when not being used for these purposes, it is available for use as a general purpose register – the Catalina Compiler makes use of it to perform various address calculations.
- BA** The **Base Address** is a 32 bit pointer initialized by the Catalina startup code to point to the base address of all other addresses in the image. This is used to locate the Catalina program in the binary image.
- BZ** The **Base End** is a 32 bit pointer initialized by the Catalina startup code to point to the end of all the static segments in the image, and the beginning of the dynamic data segment that is used for the stack and the heap.
- CS** The **Code Segment** is a 32 bit pointer containing the address of the static code segment when that segment is relocated to XMM RAM. The code segment can end up anywhere in XMM, and (once relocated) the **BA** (Base Address) is no longer sufficient to correct the internal memory references. This register is not used by the LMM Kernel.

Primitives

Each Kernel primitive is implemented as a JMP instruction to a low (and fixed) address inside the cog that is executing the kernel. Therefore, like all normal PASM instructions, each LMM primitive occupies a 32 bit long. However, unlike normal PASM instructions, many of the LMM primitives actually occupy *two* 32 bit longs, with a 32 bit long parameter following immediately after the JMP instruction.

The following table describes each primitive (in alphabetical order). The primitives marked with * require the immediately following long to contain the parameter value described in the text:

Note that unless otherwise specified, the primitives do not affect the Propeller C or Z flags.

BR_A *	This instruction (<i>Branch if Above</i>) loads the PC with the value of its 32 bit <i>address</i> parameter if and only if the Propeller's Z flag is not set and C flag is not set . BR_A automatically performs any necessary address translation.
BR_B *	This instruction (<i>Branch if Below</i>) loads the PC with the value of its 32 bit <i>address</i> parameter if and only if the Propeller's C flag is set . BR_B automatically performs any necessary address translation.
BR_Z *	This instruction (<i>Branch if Zero</i>) loads the PC with the value of its 32 bit <i>address</i> parameter if and only if the Propeller's Z flag is set . BR_Z automatically performs any necessary address translation.
BRAE *	This instruction (<i>Branch if Above or Equal</i>) loads the PC with the value of its 32 bit <i>address</i> parameter if and only if the Propeller's C flag is not set . BRAE automatically performs any necessary address translation.
BRBE *	This instruction (<i>Branch if Below or Equal</i>) loads the PC with the value of its 32 bit <i>address</i> parameter if and only if the Propeller's Z flag is set or C flag is set . BRBE automatically performs any necessary address translation.
BRNZ *	This instruction (<i>Branch if Non-Zero</i>) loads the PC with the value of its 32 bit <i>address</i> parameter if and only if the Propeller's Z flag is not set . BRNZ automatically performs any necessary address translation.
CALA *	This instruction (<i>Call Address</i>) saves the address of the <i>next instruction following CALA</i> on top of the stack (decrementing SP by 4), then loads its parameter into PC . CALA automatically performs any necessary address translation on this parameter.
CALI	This instruction (<i>Call Indirect</i>) saves the address of the <i>next instruction following CALI</i> on top of the stack (decrementing SP by 4), then loads RI into PC .
CPYB *	This instruction (<i>Copy Bytes</i>) copies a multi-byte structure from the address specified in r1 to the address specified in r0 . The parameter following this instruction contains the number of bytes to be copied. Note that r0 and r1 are not preserved. Note that CPYB destroys the Propeller Z flag.
DIVS	This instruction (<i>Division - Signed</i>) performs signed division. The 32 bit dividend must be in r0 and the 32 bit divisor must be in r1 . On return, the 32 bit quotient is in r0 and the 32 bit remainder is in r1 . Note that DIVS destroys the Propeller C and Z flags.
DIVU	This instruction (<i>Division - Unsigned</i>) performs unsigned division. The 32 bit dividend must be in r0 and the 32 bit divisor must be in r1 . On return, the 32 bit quotient is in r0 and the 32 bit remainder is in r1 . Note that DIVU destroys the Propeller C and Z flags.

FADD	This instruction (<i>Floating Point Addition</i>) performs 32 bit floating point addition. On entry, r0 and r1 contain the 32 bit numbers to be added. On return, r0 contains the result (i.e. r0 + r1).
FCMP	This instruction (<i>Floating Point Comparison</i>) performs 32 bit floating point comparison. On entry, r0 and r1 contain the 32 bit numbers to be compared. On return, the Propeller's Z flag and C flag are set.
FDIV	This instruction (<i>Floating Point Division</i>) performs 32 bit floating point division. On entry, r0 and r1 contain the 32 bit numbers to be divided. On return, r0 contains the result (i.e. r0 / r1).
FMUL	This instruction (<i>Floating Point Multiplication</i>) performs 32 bit floating point multiplication. On entry, r0 and r1 contain the 32 bit numbers to be multiplied. On return, r0 contains the result (i.e. r0 * r1).
FSUB	This instruction (<i>Floating Point Subtraction</i>) performs 32 bit floating point subtraction. On entry, r0 and r1 contain the 32 bit numbers to be subtracted. On return, r0 contains the result (i.e. r0 - r1).
FLIN	This instruction (<i>Floating Point from Integer</i>) converts the integer in r0 to a floating point value. On return, the result is in r0 .
INFL	This instruction (<i>Integer from Floating Point</i>) converts the floating point value in r0 to an integer. On return, the result is in r0 .
INIT	This instruction (<i>Initialization</i>) is the main entry point for the kernel. It is called only once, on startup.
JMPA *	This instruction (<i>Jump Address</i>) loads the PC with the value of its 32 bit parameter. JMPA automatically performs any necessary address translation.
JMPI	This instruction (<i>Jump Indirect</i>) loads the PC with the value of RI .
LODA *	This instruction (<i>Load Address</i>) loads its 32 bit <i>address</i> parameter into RI . The difference between LODA and LODL is that LODA treats its parameter as an address, and automatically performs any necessary address translation.
LODF *	This instruction (<i>Load Frame Reference</i>) loads its 32 bit signed offset parameter into RI , then adds FP to it. When executed within a function, this means that RI will contain either the address of a local variable (negative offset) or one of the arguments to the function (positive offset).
LODL *	This instruction (<i>Load Long</i>) loads its 32 bit parameter into RI . The difference between LODL and LODA is that LODL does not perform any address translation.
MULT	This instruction (<i>Multiplication</i>) performs multiplication. The 32 bit multiplicand must be in r0 and the 32 bit multiplier must be in r1 . On return, the 32 bit result is in r0 . Note that MULT destroys the Propeller C and Z flags.

POPM *	This instruction (<i>Pop Many Registers</i>) treats its 32 bit parameter as a bitmap specifying which of the general purpose registers to pop from the stack. E.g. if bit 23 is set, r23 is popped (decrementing SP by 4), then r22 etc ... down to r0 . Note that POPM destroys the Propeller C flag.
PSHA *	This instruction (<i>Push Address</i>) pushes its 32 bit <i>address</i> parameter into the stack (decrementing SP by 4). PSHA automatically performs any necessary address translation.
PSHB *	This instruction (<i>Push Bytes</i>) pushes a multi-byte structure onto the stack. The structure to be pushed must be pointed to by r0 . The parameter following this instruction contains the number of bytes to be pushed. SP is decremented by the number of bytes, rounded up to the next multiple of 4. Note that r1 is not preserved. Note that PSHB destroys the Propeller C and Z flags.
PSHF	This instruction (<i>Push Frame Reference</i>) loads its 32 bit signed offset parameter into RI , then adds FP to it. It then uses that value as an address, and pushes the value found at that address onto the stack (decrementing SP by 4). The result is that the top of the stack ends up with the value of either a local variable (negative offset) or one of the arguments to the function (positive offset).
PSHL	This instruction (<i>Push Long</i>) pushes the 32 bit contents of RI onto the stack (decrementing SP by 4).
PSHM *	This instruction (<i>Push Many Registers</i>) treats its 32 bit parameter as a bitmap specifying which of the general purpose registers to push on the stack. E.g. if bit 0 is set, r0 is pushed (incrementing SP by 4), then r1 etc ... up to r24 . Note that PSHM destroys the Propeller C flag.
NEWF	This instruction (<i>New Frame</i>) saves the current value of FP on the stack (decrementing SP by 4), and sets up a new frame pointer in FP , allocating BC bytes for local storage. If this instruction is executed as the first instruction of a function, then on exit BC contains the value of SP before the function was called – this assists in accessing arguments to the function that have been pushed onto the stack.
RETF	This instruction (<i>Return from Frame</i>) discards the current frame pointer by loading FP with the value on the top of the stack (incrementing SP by 4). It then loads the PC with the value on the top of the stack (incrementing SP by 4).
RBYT	This instruction (<i>Read Byte</i>) loads the low order byte of the BC register with the value of the byte in memory pointed to by RI . The remainder of the BC register will be zero. See the note below on Kernel Memory Models for details on which memory is accessed by this instruction.
RLNG	This instruction (<i>Read Long</i>) loads the low order word of the BC register with the value of the long in memory pointed to by RI . The remainder of

	the BC register will be zero. See the note below on Kernel Memory Models for details on which memory is accessed by this instruction.
RWRD	This instruction (<i>Read Word</i>) loads the BC register with the value of the word in memory pointed to by RI . See the note below on Kernel Memory Models for details on which memory is accessed by this instruction.
RETN	This instruction (<i>Return</i>) loads the PC with the value on the top of the stack (incrementing SP by 4). This is only used by trivial functions that do not use NEWF – if NEWF has been called, RETF should be used instead.
SYSP	This instruction (<i>System Plugin</i>) invokes an external plugin. On entry, r2 contains either the cog that contains the plugin (bit 7 set) or the type of plugin to be called (bit 7 not set), and r3 contains the data to send to the plugin. On return, r0 will contain the result of the call, or -1 if the plugin could not be located. Note that SYSP destroys the Propeller C and Z flags.
WBYT	This instruction (<i>Write Byte</i>) writes the low order byte of the BC register to the byte of memory pointed to by RI . See the note below on Kernel Memory Models for details on which memory is accessed by this instruction.
WLNG	This instruction (<i>Write Long</i>) writes the BC register to the long of memory pointed to by RI . See the note below on Kernel Memory Models for details on which memory is accessed by this instruction.
WWRD	This instruction (<i>Write Word</i>) writes the low order word of the BC register to the word of memory pointed to by RI . See the note below on Kernel Memory Models for details on which memory is accessed by this instruction.

Kernel Memory Models

The RLNG, RWRD, RBYT, WLNG, WWRD and WBYT primitives are used to write to Hub RAM and/or XMM RAM, depending on the kernel and memory model in use:

LMM Kernel:	These instructions only access Hub RAM.
XMM Kernel (small):	These instructions only access XMM RAM.
XMM Kernel (large):	These instructions can be used to access both XMM RAM and Hub RAM.

Unsupported PASM

The LMM and XMM Kernels cannot be used to execute arbitrary PASM instructions. Attempting to execute some PASM instructions would disrupt the operation of the kernel itself. The Catalina Code Generator never generates such instructions – instead, where necessary, it generates the code required to invoke the equivalent LMM primitive instead.

The following table summarizes the PASM instructions that should not be used within programs intended to be executed by the Catalina Kernel, and also (where appropriate) the LMM equivalent that should be substituted instead:

PASM	LMM Equivalent
Conditional JMP	BR_Z, BRNZ, BR_A, BRAE, BR_B, BRBE
Unconditional JMP	JMPA (or load the address in RI and use JMPI)
RET	RETN (or RETF if NEWF has been called)
JMPRET or CALL	CALA (or load the address in RI and use CALI)
DJNZ	SUB and then BRNZ
TJZ	CMP and then BR_Z
TJNZ	CMP and then BRNZ

Object and Image Formats

By default, Catalina executables use the same **binary** or **eeeprom** executable image format as other Propeller executables (although note that other formats are available – see the Catalina documentation).

However, only Catalina programs compiled as LMM programs (i.e. using *layout 0*, specified using the **-x0** option, or when no other layout is specified) can be easily loaded and run on a Propeller, since only those are of size 32kb or less.

EMM executables (using *layout 1*, specified using the **-x1** option) will always be between 32 kb and 64 kb, and “small” XMM executables (using *layout 2*, specified using the **-x2** option) or “large” XMM executables (using *layout 5*, specified using the **-x5** option) will always be larger than 32 kb – and can in fact be any size.

We will deal with Catalina LMM images first, as they are the simplest. In an LMM executable image, the compiled Catalina C program is contained within a normal chain of SPIN objects – which will include the Catalina Target, the various plugins that target is responsible for loading, and the LMM Kernel. The compiled Catalina C program object hence has a base address in the same way that every SPIN object has a base address.

In addition, the Catalina C program object has two key items of internal structure, at the offsets specified in the file **Catalina_Common.spin**:

```
LMM_INIT_BZ_OFF = $4b
LMM_INIT_PC_OFF = $4c
```

As the names imply, these are the offsets (in longs) from the base address (+8) within each object of the initial values of **BZ**, and the initial values of the **PC** (which in C will always be the address of the *main* function). These values must be extracted from within the object itself.

The initial value of **BA** is the object's base address, and the initial value of the **SP** is always the top of memory just below any RAM space allocated to the plugins.

Catalina expects to be able to use the memory following **BZ** as dynamic data (i.e. heap or stack space). But the initial **BZ** offset is not only the pointer to the *start* of the dynamic data segment - it also points to the *end* of a special structure consisting of 5 longs, as follows:

```

CODE   Start of static code segment
CNST   Start of static constant data segment
INIT   Start of static initialized data segment
DATA   Start of static uninitialized data segment
ENDS   End of all static segments

```

This structure contains values that would need to be used by the kernel when relocating the various program segments. In fact, the LMM Kernel does not relocate any segments, and therefore does need to use any of these extra values - they are only needed by the XMM Kernel.

The EMM targets differ from the LMM targets in that they *don't* include the compiled Catalina C program objects – but these targets don't need to relocate individual segments either. The EMM targets are self-contained SPIN programs that occupy the first 32kb of external EEPROM. Instead of containing the Catalina C program object, the EMM targets expect to load this object from address \$8000 in the external EEPROM (i.e. in the second 32kb). Once all the target plugins have been loaded, the EMM loader simply overwrites the whole of hub RAM with the Catalina C program object found at that address – and in the process, overwrites itself with the Catalina LMM Kernel. The only value the EMM targets therefore have to manipulate is the overall Base Address (**BA**).

The situation is slightly more complex for XMM targets. The current XMM targets are also self-contained SPIN programs that expect to load the compiled Catalina C program object stored at location \$8000 in the external EEPROM – but the XMM targets must use the 5 long structure described above to decompose the compiled Catalina C program object – currently the static code segments are moved to XMM RAM and the static data segments are moved to hub RAM. Thus the XMM targets currently need to set up the **CS** register, and (in future, when the static and dynamic data segment may also move to XMM RAM) they may also need to set up the **SD** and **DD** registers.

Catalina Calling Conventions

Although only indirectly related to the LMM or XMM kernels themselves, it is worth spending some time discussing the function calling conventions adopted by the Catalina Code Generator, since the primitives implemented by the Catalina kernels have been optimized to support this particular type of calling convention.

The most complex part of the Code Generator is concerned with the setup of the arguments (and the local variables) on the stack when calling a function. These

conventions must be understood and adopted by any PASM function intended to be directly callable from Catalina.

In order to make most effective use of the scarce Propeller resources, such as the limited cog RAM (in the form of the general purpose registers) as well as the limited hub RAM, and also to minimize the overhead of performing each function call, Catalina adopts the following calling conventions:

- The result of a function is always returned in **r0**.
- The caller must clean up the stack. This means that if a function expects parameters, the caller must not only push the parameters onto the stack before making the call, it must adjust the stack again afterwards when the call is complete. The called procedure simply uses the parameters provided and then returns its result in **r0**.
- The parameters are always passed in reverse order. For example, when calling **function(a,b,c,d,e)** – and assuming all the parameters are passed on the stack – they must be pushed onto the stack in order: **e, d, c, b, a**.
- Up to 4 of the formal parameters to a function can be passed in registers **r2**, **r3**, **r4**, and **r5**. Given that parameters are processed in reverse order, this is actually the *last* four parameters. For example, when calling a function such as **function(a,b,c,d,e)** it would mean **e** is passed in **r2**, **d** is passed in **r3**, **c** is passed in **r4**, and **b** is passed in **r5**. Parameter **a** in such cases would need to be pushed onto the stack – which would also have space allocated for parameters **b**, **c**, **d** and **e** even though it is not used unless the argument is actually a structure, or the function is *variadic*, or the function takes the address of the argument – in such cases the register passing is not possible, and the corresponding register is not used.

These calling conventions may seem (and in fact are) quite complex to implement. But they have the advantage that probably 75% (or more) of all function calls – especially so-called “*lea*” functions that do not themselves make any further function calls – can simply be made by loading the arguments into the appropriate registers and then calling the function – without having to first push each argument onto the stack. Similarly, in most cases the function itself can simply use the arguments it finds in the registers – without needing to load them from the stack.

In a recursive program, or one that consists of a large number of relatively trivial function calls, this calling convention saves large amounts of program code – and also make the resulting program much faster to execute.

A Description of the Standard Catalina XMM API

Supporting new XMM hardware is quite simple with Catalina – the XMM Kernel can be customized to include new XMM hardware access routines, or (if those routines will not fit in the Kernel) then the cache support option can be used instead.

All newly developed XMM access routines must conform to the following general XMM API, so that the core parts of the Catalina Kernel remain identical on all supported platforms.

Note that it is important that (unless otherwise specified) ***no XMM API routine affects the processor flags***, such as the Z or C flags. If these flags must be used, they should be saved and restored by the XMM API functions.

The XMM API now consists of three distinct parts:

The *cache* access functions – these are mandatory in all cases. If only these functions are implemented, they must fit in the space available in the cache cog – about 355 longs.

The *direct* access functions – these are optional. However, without these functions, only cached access will be supported. If implemented, these functions must fit in the available space in the kernel cog - about 96 longs.

The *flash* access functions – these are mandatory for FLASH support. FLASH always requires the use of the cache. If only these are implemented, they and the cache access functions must fit in the space available in the cache cog – about 355 longs.

The XMM API cache access functions

The cache access functions are the minimum XMM functions required. These functions are used by the Caching XMM driver (enabled when one of the CACHED symbols is defined). The advantage of the caching XMM driver is that it only requires four functions to be implemented:

XMM_Activate - This routine is called during XMM Kernel initialization to initialize the XMM Hardware. It may also be called (if configured for the platform by defining the symbol **XMM_SHARED**) at the conclusion of the execution of any system plugin call – see also **XMM_Tristate**.

On entry:

None.

On exit:

None.

XMM_Tristate - This routine can be called ((if configured for the platform by defining the symbol **XMM_SHARED**)) at the beginning of any system plugin call – this is needed if a plugin may need to use hardware that cannot be used while the XMM memory hardware is active. If this is the case, the XMM

hardware will be reactivated by calling **XMM_Activate** at the conclusion of the system plugin call.

On entry:

None.

On exit:

None.

XMM_ReadPage - Read multiple bytes from XMM RAM to Hub RAM.

On entry:

XMM_Addr destination address in XMM RAM

Hub_Addr Destination address in main memory (16-bits used)

XMM_Len number of bytes to read.

On exit:

XMM_Addr incremented by the number of bytes read.

XMM_WritePage - Write multiple bytes from Hub RAM to XMM RAM.

On entry:

XMM_Addr destination address in XMM RAM

Hub_Addr source address in main memory (16-bits used)

XMM_Len number of bytes to write.

On exit:

XMM_Addr incremented by the number of bytes written.

The XMM API direct access functions

The direct access functions are additional functions required to support direct XMM access from within the XMM kernel. They are optimized to provide the type of access the Kernel needs, and generally provide the best performance. However, it is sometimes not possible to fit the direct functions in the Kernel. In that case, the cache has to be used to access the XMM RAM.

Direct access requires the cache access functions, plus four additional functions:

XMM_ReadLong - Read a long from XMM RAM to Cog RAM. Note that this function is used for all instruction fetches, and should be optimized for speed if possible.

On entry:

XMM_Addr XMM address to read (the number of bits used depends on the platform)

XMM_Dst The destination of this instruction must be set to the destination address in Cog RAM.

On exit:

XMM_Addr incremented by 4.

Destination Cog RAM location that will contain the long read from XMM.

XMM_ReadMult - Read multiple bytes from XMM RAM to Cog RAM. Bytes are read into the destination location least significant byte first, so this routine can be used to read byte, word or long values.

On entry:

XMM_Addr XMM address to read (the number of bits used depends on the platform)

XMM_Dst destination of this instruction set to destination address in Cog RAM.

XMM_Len number of bytes to read (1, 2 or 4).

On exit:

XMM_Addr incremented by the number of bytes read.

Destination Cog RAM location that will contain the bytes read from XMM.

XMM_WriteLong - Write a long from XMM RAM to Cog RAM.

On entry:

XMM_Addr XMM address to write (the number of bits used depends on the platform)

XMM_Src The source of this instruction must be set to the source address in Cog RAM.

On exit:

XMM_Addr incremented by 4.

Source Cog RAM location that contains a long value to be written to XMM.

XMM_WriteMult - Write multiple bytes from Cog RAM to XMM. Bytes are written from the source location least significant byte first, so this routine can be used to write byte, word or long values.

On entry:

XMM_Addr XMM address to write (the number of bits used depends on the platform).

XMM_Src The source of this instruction must be set to the source address in Cog RAM.

XMM_Len number of bytes to write (1, 2 or 4).

On exit:

XMM_Addr incremented by the number of bytes read.

Source Cog RAM location that contains the bytes to be written to XMM.

The XMM API flash access functions

Some platforms implement XMM RAM using FLASH, or a combination of FLASH and SRAM. If any FLASH RAM is used, the cache must be used, and following additional 11 functions are required:

XMM_FlashActivate - Called to initialize the Flash hardware (if required). Similar to **XMM_Activate**.

On entry:

None.

On exit:

None.

XMM_FlashTristate - Called to disable the Flash hardware (if required). Similar to **XMM_Tristate**.

On entry:

None.

On exit:

None.

XMM_FlashWritePage - Write multiple bytes of Hum RAM to FLASH. For some flash RAM, writes are limited to a maximum of 256 bytes.

On entry:

XMM_Addr destination address in FLASH RAM.

Hub_Addr Source address in main memory (16-bits used).

XMM_Len number of bytes to write (up to 256).

On exit:

XMM_Addr incremented by the number of bytes written.

XMM_FlashReadPage - Read multiple bytes from FLASH to Hub RAM.

On entry:

XMM_Addr destination address in FLASH RAM

Hub_Addr Destination address in main memory (16-bits used)

XMM_Len number of bytes to read.

On exit:

XMM_Addr incremented by the number of bytes read.

XMM_FlashComparePage - Compare multiple bytes of Hub RAM with FLASH. Used if the symbol WRITE_CHECK is defined.

On entry:

XMM_Addr address in FLASH RAM

Hub_Addr address in main memory (16-bits used)

XMM_Len number of bytes to compare.

On exit:

Z Flag set if equal, cleared if not equal.

XMM_Addr incremented by the number of bytes equal.

Hub_Addr incremented by the number of bytes equal.

outx Value of last main memory (Hub) byte compared.

XMM_FlashCheckEmpty - Check that a section of FLASH is empty (all 0xFF). Used if the symbol ERASE_CHECK is defined.

On entry:

XMM_Addr address in FLASH RAM

XMM_Len number of bytes to check.

On exit:

Z Flag set if all empty, cleared if not empty.

XMM_Addr incremented by number of bytes compared.

outx Value of last FLASH byte checked.

XMM_FlashEraseChip - Erase the entire FLASH. Used by loaders if the CHIP_ERASE symbol is defined.

On entry:

None.

On exit:

None.

XMM_FlashEraseBlock - Erase a 4k block of FLASH. Used by loaders if the CHIP_ERASE symbol is *not* defined (the default). If the FLASH chip cannot erase 4k blocks, this may be a null routine (and CHIP_ERASE must be defined).

On entry:

XMM_Addr address in FLASH RAM of 4k block to erase.

On exit:

None.

XMM_FlashUnprotect - Unprotect the entire FLASH. Some FLASH chips require the chip to be unprotected before any Write operations (and sometimes before any Read operations). If this is not required, this may be a null routine.

On entry:

None.

On exit:

None.

XMM_FlashWriteEnable - Enable Writing to the FLASH. Some FLASH chips require a write enable to be performed before each Write operation. If this is not required, this may be a null routine.

On entry:

None.

On exit:

None.

XMM_FlashWaitUntilDone - Loop until a Write or Erase operation completes.

On entry:

None.

On exit:

outx Contents of the FLASH status register.

Other XMM support routines may be required, especially if the XMM platform uses memory paging – a typical implementation might (for example) also store a current page register, and provide routines to set up and/or increment XMM addresses that takes the current page into account. However, such routines are platform dependent and do not form part of the standard XMM API.

A Description of the Catalina Addressing Modes

On the Propeller 1, Catalina supports four different addressing modes:

- | | |
|--------------|--|
| TINY | In this mode, all addresses are Hub RAM addresses – this means that all code and data addresses must be less than 32k (at least on the Propeller 1 - on the Propeller 2 this is 512k). This is the mode used by the LMM Kernel for all programs compiled using the -x0 or -x1 or -C TINY command line option. |
| SMALL | In this mode, all data addresses are Hub RAM addresses, but all code addresses are XMM RAM addresses. Hub RAM addresses |

must be less than 32k, but code addresses can be up to 16 Mb. This mode is used by the XMM Kernel, for programs compiled using the **-x2** or **-C SMALL** command line option.

LARGE In this mode, all data and code addresses are XMM addresses, and can be up to 16 Mb. But at run time, all stack and frame addresses are Hub addresses, and must be less than 32k. This includes the addresses of all local variables, which are constructed “on the fly” on the stack. This mode is used by the XMM Kernel, for programs compiled using the **-x5** or **-C LARGE** command line option.

COMPACT In this mode, all addresses are Hub RAM addresses – however, all code and data addresses are specified using 24 bits, making this mode suitable for both the Propeller 1 and the Propeller 2. This is the mode used by the CMM Kernel for all programs compiled using the **-x8** or **-C COMPACT** command line option.

The **TINY** memory model is conceptually the simplest, since all addresses are Hub RAM addresses. However, it can only be used for programs where everything (i.e. the program code, data, heap and stack) can fit into Hub RAM.

The **SMALL** memory model is more complex. The Kernel has to accommodate the differences between Hub RAM and XMM RAM. However, passing data around between different plugins is still relatively simple, since all data addresses are Hub RAM addresses.

The **LARGE** memory model is actually simpler internally than the Small model, but there is one complication – in this mode data addresses can be either an XMM RAM address (e.g. a heap address) or a Hub RAM address (e.g. a local variable address). Both are equally valid at the program level, but XMM RAM addresses cannot be used to pass data between plugins. This means, for example, that it is not possible to pass the address of a static string (which will be an XMM RAM address) directly to a plugin (such as a HMI plugin). Catalina works around this limitation by copying data from XMM RAM to Hub RAM whenever it must be passed to a plugin. At the program level, this is invisible to the user, but it has implications when writing plugins or other PASM code to be executed in conjunction with Catalina programs.

The **COMPACT** memory model is similar to the **TINY** memory model – but the code itself is completely different, being a hybrid of LMM code and interpreted code.

A Description of the Catalina Propeller 1 Image Format

All Catalina Propeller 1 program images consist of 2 parts – a SPIN program that establishes the environment for the program by loading the plugins, the drivers and the Kernel itself, and then the compiled program that the Kernel will execute once loaded.

The compiled program image executed by the Kernel code always starts with the same ‘prologue’, as follows:

```
' Catalina programs all start at offset 0
    org      0
' the first 2 longs are reserved (one of them is required by the POD
' POD debugger when it is used to debug LMM programs)
    long     0          '$00
    long     0          '$01
' the next 41 longs contain JMP instructions for each Kernel primitive
' (the actual values not significant in the program image - only the
' address of each jump instruction is important)
INIT      jmp     0          '$02
LODL      jmp     0          '$03
LODA      jmp     0          '$04
LODF      jmp     0          '$05
PSHL      jmp     0          '$06
PSHB      jmp     0          '$07
CPYB      jmp     0          '$08
NEWF      jmp     0          '$09
RETF      jmp     0          '$0a
CALA      jmp     0          '$0b
RETN      jmp     0          '$0c
CALI      jmp     0          '$0d
JMPA      jmp     0          '$0e
JMPI      jmp     0          '$0f
DIVS      jmp     0          '$10
DIVU      jmp     0          '$11
MULT      jmp     0          '$12
BR_Z      jmp     0          '$13
BRNZ      jmp     0          '$14
BRAE      jmp     0          '$15
BR_A      jmp     0          '$16
BRBE      jmp     0          '$17
BR_B      jmp     0          '$18
SYSP      jmp     0          '$19
PSHA      jmp     0          '$1a
FADD      jmp     0          '$1b
FSUB      jmp     0          '$1c
FMUL      jmp     0          '$1d
FDIV      jmp     0          '$1e
FCMP      jmp     0          '$1f
FLIN      jmp     0          '$20
INFL      jmp     0          '$21
PSHM      jmp     0          '$22
POPM      jmp     0          '$23
PSHF      jmp     0          '$24
RLNG      jmp     0          '$25
RWRD      jmp     0          '$26
```

```

RBYT      jmp      0          '$27
WLNG      jmp      0          '$28
WWRD      jmp      0          '$29
WBYT      jmp      0          '$2a
' the next 8 longs contain the internal Kernel registers
PC         long     0          '$2b
SP         long     0          '$2c
FP         long     0          '$2d
RI         long     0          '$2e
BC         long     0          '$2f
BA         long     0          '$30
BZ         long     0          '$31
CS         long     0          '$32
' the next 24 longs contain the general purpose registers
r0         long     0          '$33
r1         long     0          '$34
r2         long     0          '$35
r3         long     0          '$36
r4         long     0          '$37
r5         long     0          '$38
r6         long     0          '$39
r7         long     0          '$3a
r8         long     0          '$3b
r9         long     0          '$3c
r10        long     0          '$3d
r11        long     0          '$3e
r12        long     0          '$3f
r13        long     0          '$40
r14        long     0          '$41
r15        long     0          '$42
r16        long     0          '$43
r17        long     0          '$44
r18        long     0          '$45
r19        long     0          '$46
r20        long     0          '$47
r21        long     0          '$48
r22        long     0          '$49
r23        long     0          '$4a
' the next 6 longs contain some constants that are required either in
' the Kernel or in the compiled program - while there are many others
' that could have been included here, these ones are specifically
' required for various purposes
Bit31      long     $80000000   '$4b
all_1s     long     $ffffffff   '$4c
cviu_m1    long     $000000ff   '$4d
cviu_m2    long     $0000ffff   '$4e
top8       long     $ff000000   '$4f  ' top 8 bits bitmask
low24      long     $00ffffff   '$50  ' low 24 bits bitmask
' the next 2 longs contain initial program values
init_BZ    long     @sbrkinit   '$51  ' end of code / start of heap
init_PC    long     @C_main     '$52  ' the initial PC
' the next long contains the segment layout (i.e. 0, 1, 2, 3, 4 or 5)
seglayout  long     SEGMENT_LAYOUT
' the next 4 longs contains the start address of each of the segments,
' followed by a long containing the address of the first byte after
' all the segments

```

```

segtable    long    @Catalina_Code
            long    @Catalina_Cnst
            long    @Catalina_Init
            long    @Catalina_Data
            long    @Catalina_Ends
            long    @Catalina_RO_Base
            long    @Catalina_RW_Base

```

In the program image, this prologue is then followed by each of the segments themselves (i.e. the **Code**, **Cnst**, **Init** and **Data** segments). The address of each segment is given in **segtable**, and the segment layout and address mode is specified in the **seglayout**.

It is important to note that the first 83 longs (i.e. longs \$00 to \$52) are identical between the program image and the Kernel – i.e. longs with the same names are present within both the compiled program image and within the Kernel. In most cases, the actual *values* contained in these longs in the program image are not significant – the exceptions are the **init_BZ**, **init_PC**, **seglayout** and **segtable** values - the kernel must retrieve these from the program image before program execution can begin.

Beyond the first 83 longs (i.e. beyond longs \$00 to \$52) the values in the prologue above are present in the image, but equivalent long values do not exist within the Kernel. Additional longs beyond those shown (i.e. after the **segtable** entries, but before any actual program segments) may be present in the program image, but the prologue is guaranteed never to exceed 512 bytes (i.e. 128 longs, or longs \$00 to \$7F).

The remaining details of the image format used by a Catalina program depends on both the memory model (i.e. LMM, EMM or XMM) and the addressing mode (i.e. TINY, SMALL, LARGE or COMPACT) selected during compilation – these are controlled by the **-x** command line parameter. The following are the main differences in the image format for each supported value of this parameter:

- x0** The image is a normal Propeller format image (binary or eeprom). The total size of the image (i.e. the Kernel, plus all the required plugins and drivers, plus the compiled program) must be 32k or less. The compiled program embedded in the image must be executed by an LMM Kernel that uses the **TINY** addressing mode – this means that all code and data addresses are Hub addresses. This image format can be loaded by any means supported for normal SPIN programs, as well as by the Catalina Generic SD Loader.
- x1** The image consists of two sections – the first 32k is a normal SPIN program that contains an LMM kernel, plus all the required plugins and drivers, and also a loader that knows how to load a compiled program from the second 32k stored in an external EEPROM. The compiled program embedded in the image must be executed by an LMM mode Kernel that uses the **TINY** addressing mode – this means that all code

and data addresses are Hub addresses. This image format cannot be loaded by normal SPIN tools, and is specifically intended to be programmed into a 64kb EEPROM.

- x2 The image consists of two sections – the first 32k is a normal SPIN program that contains a **SMALL** mode XMM kernel plus all the required plugins and drivers, and also a loader that knows how to load a compiled program from XMM – which is where the Catalina Generic Program Loader puts anything after the first 32k when it loads such programs. The compiled program embedded in the image must be executed by an XMM Kernel that uses the “Small” addressing mode – this means that all code addresses are XMM addresses, but all data addresses are Hub addresses. This image format cannot be loaded by normal SPIN tools, but can be loaded using the Catalina Generic SD Loader.
- x5 The image consists of two sections – the first 32k is a normal SPIN program that contains a **LARGE** mode XMM kernel plus all the required plugins and drivers, and also a loader that knows how to load a compiled program from XMM – which is where the Catalina SD Program Loader puts anything after the first 32k when it loads a program. The compiled program embedded in the image must be executed by an XMM Kernel that uses the **LARGE** addressing mode – this means that all code, data and heap addresses are XMM addresses, but all stack and frame addresses are Hub addresses. This image format cannot be loaded by normal SPIN tools, but can be loaded using the Catalina Generic SD Loader.
- x8 Similar to -x0, except the code is not LMM code – it is CMM code, which is a hybrid of LMM and interpreted code.
- x9 Similar to -x1, except the code is not LMM code – it is CMM code, which is a hybrid of LMM and interpreted code.

As implied above, there are several ways to load Catalina program images, depending on the image format:

- x0 These programs can be loaded using any standard Propeller tool, and can also be loaded using the Catalina Generic SD Loader, or the Catalina Payload utility.
- x1 These programs cannot be “loaded” as such – they must be programmed into an EEPROM. The mechanism for doing this is platform dependent.
- x2 These programs must be loaded using the Catalina Generic SD Loader, or the Catalina Payload program.
- x5 These programs must be loaded using the Catalina Generic SD Loader or the Catalina Payload program.
- x8 These programs must be loaded using the Catalina Payload program.

- x9 These programs cannot be “loaded” as such – they must be programmed into an EEPROM. The mechanism for doing this is platform dependent.

A description of the Generic SD Loader is given in the next section.

A Description of the Propeller 1 Generic SD Loader

Now that Catalina has SD Card file system support, it made sense to include an SD Loader that can be used to load Catalina programs from an SD Card.

However, the file system support mandated by ANSI C (and as implemented by Catalina) is quite complex and is really intended for Propeller platforms that also have XMM memory support. Even simple programs that use the Catalina SD file system will be larger than 32k - this means the ANSI C file system is not suitable for use by an SD Loader that has to fit into 32k.

But while the SD Loader has to be small, it does not need to be particularly fast - so instead of using C, the Catalina Loader is implemented as a SPIN program which uses the FATEngine file system module. This means that the SD Loader is essentially independent of Catalina and can be used to load normal SPIN programs from the SD Card as well as Catalina C programs.

The major feature of the Catalina SD Loader is that in addition to being able to load normal SPIN or LMM C programs (or any other program which fit within the Propeller's 32k of Hub RAM) the Loader also knows how to load programs and/or data into XMM RAM. This allows programs with code segments up to 16 Mb to be compiled on a PC, and then loaded into the Propeller via an SD Card.

The SD Loader currently loads programs based only on the size of the file being loaded. Files of 32kb or less (actually 31k, since the loader requires 1k for its internal SD card buffers) are loaded into Hub RAM. Files larger than 32k are assumed to consist of two parts - the first 31k is loaded into Hub RAM, while anything beyond the 32k boundary is loaded into XMM RAM.

Other than requiring the use of 1kb during the load process, the SD Loader imposes no other limitations or overheads - i.e. it does not consume any cogs or RAM space once the program has been loaded and begins executing.

Although this sounds simple enough, the details of the whole process of booting a program from an SD Card are quite complex.

Here is an overview of the process:

Phase I: The SD Loader loads the file sector list of the selected file into Hub RAM, and then starts the Sector Loader.

Phase II: The Sector Loader loads the file sectors themselves into both Hub RAM and XMM RAM, then starts the Target program in Hub RAM.

In multi-CPU systems where the program is to be loaded into another CPU, the SD loader does not load the sectors into local RAM – instead, it sends each sector via serial I/O to the target CPU, and the subsequent phases are executed on the target CPU.

For SPIN or LMM C programs, the process ends here. For XMM C programs the process then carries on

Phase III: The Target program loads the Catalina Plugins into Cog RAM, and then starts the Hub Loader.

Phase IV: The Hub Loader copies the XMM Kernel from Hub RAM to XMM RAM, then loads the Catalina data segments from XMM RAM to Hub RAM, then copies the XMM Kernel back to Hub RAM from XMM RAM. Finally, the Hub Loader starts the XMM Kernel.

And here are all the details:

Phase I:

1. The SD Loader uses **FATEngine** to mount the SD card, determine the sector geometry of the card, and also to load the root directory of the card.
2. The SD Loader can be configured to either load a specific file, or display the root directory of the SD Card and allow the user to select one. If it is configured for the latter, the SD Loader loads and starts an instance of a tv and keyboard driver, and displays the root directory of the SD Card, allowing the user to select the name of a file to be loaded.
3. The cluster list of the selected file is loaded into RAM (at the fixed address of \$7F00). This is required because **FATEngine** (which is a normal SPIN program running from Hub RAM) must be terminated before the selected file can be loaded - but once **FATEngine** is terminated the SD Card can only be interrogated using "low level" sector-based I/O - not "high level" file-based I/O.
4. If the SD Loader started a keyboard and tv driver, these are terminated once the cluster list has been successfully loaded.
5. The SD Loader then starts the Sector Loader - this is a PASM program that uses the cluster list and the sector geometry of the SD Card to load the sectors that make up the selected file.

Phase II:

1. In single CPU systems, or in multi-CPU systems where a program is to be loaded into the local CPU, the Sector Loader loads the first 31kb of the file into Hub RAM - and anything after the first 32k into XMM RAM. Note that neither the SD Loader nor the Sector Loader know how to interpret any of the data they load - they simply cooperate to load the data into Hub RAM or XMM RAM based on where it was in the file. This is why the Catalina Loader is referred to as a "Generic" Loader - there is nothing Catalina specific in any of these operations, and the Loader can be used to load anything - either program or data - into Hub RAM and XMM RAM. Of course, it assumes that the start of Hub RAM is a valid Propeller program and executes it - but the rest of the Hub RAM or the XMM RAM could contain anything.
2. In multi-CPU systems where the program is to be loaded into another CPU, the SD loader instead sends each sector via serial I/O to a **Generic SIO Loader** executing on the target CPU, which does the same job as described in the previous step. The subsequent phases are then executed on the target

CPU. Note that this loader must already be executing on the target CPU – this can be accomplished by loading it into EEPROM and having it always start on boot, or by first executing a separate CPU boot loader, which uses the built-in Parallax load capabilities to load a program from one CPU to another (the built-in Parallax loader is only suitable for loading programs up to 32k in size, which is why we need a two-step load for Catalina programs – which may be larger than 32k). The use of these utilities is fully described in the **Multi-CPU system Support** section of the **Catalina Reference Manual**

3. If it loaded a program locally, the Sector Loader then performs a "soft reset" by shutting down any other cogs and then restarting its own cog using the normal Propeller boot code - essentially, this starts the cog as a SPIN interpreter, which will begin executing whatever program was just loaded into Hub RAM - i.e. whatever program was in the first 31kb of the selected file. If it loaded a program into another CPU, the Sector Loader instead does a "hard reset" of the Propeller – which normally restarts the SD Loader.

If the program loaded is either a normal SPIN program, or a Catalina LMM C target, that's the end of the process - the loaded program just executes normally. But if the program that has just been loaded into Hub RAM (and started) is in fact a Catalina XMM target, there is still more work to do before the Catalina program can be started.

Phase III:

1. To be successfully booted from an SD Card, the Target program must know that the program to be executed has already been loaded into XMM RAM by the Sector Loader. This is why programs to be loaded from SD Card typically require a different target to programs intended to be loaded from EEPROM. Note that there is no need for a specific LMM target to use the Loader - LMM programs exist entirely within Hub RAM, and do not need to do any memory re-organization - it is irrelevant to the LMM program how it got loaded into Hub RAM.
2. Like any other target, a Target loaded from the SD Card must load into Cog RAM all the plugins required by the Catalina program to be executed - including an SD Card plugin if the Catalina program itself requires access to the SD file system.
3. The Target must also contain the XMM Kernel needed to execute the Catalina program - but this Kernel cannot be started just yet. First, the Kernel itself must be temporarily copied from Hub RAM to XMM RAM so that the Hub RAM can be re-organized. To do this, the target loads a Hub Loader, which is yet another low-level loader program that specifically know how to find the Kernel code in the target, and also how to find the data and code segments of the Catalina program that the Sector Loader has loaded into XMM RAM.

Phase IV:

1. The XMM Hub Loader moves the Kernel to XMM RAM, after the data the Sector Loader loaded from the SD Card.

2. The XMM Hub Loader moves the data segments of the Catalina Program in XMM RAM to Hub RAM.
3. The XMM Hub Loader relocates the code segment in XMM RAM so that it is where the Kernel expects to find it. While it is possible to define a new memory segment layout that would avoid the need for this step, doing so would further complicate the Catalina compilation process, so instead the XMM Hub Loader uses SD-specific targets that know how to relocate a normal Catalina x2 XMM layout.
4. The XMM Hub Loader moves the Kernel from XMM RAM to Hub RAM. It is copied into the space which (once the program is started) will be used as heap and stack space - Note that this means that all Catalina XMM programs must limit their constant data segments to 28kb.
5. Finally, the Hub XMM Loader restarts itself as the XMM Kernel, and begins executing the Catalina program in XMM.

This load process is so complex mainly because the Propeller is so resource limited – but there are also some design decisions that have led to further complications, such as the decision to use a Generic Loader that does not decode the Catalina image, instead requiring the loaded program to do this job itself. These decisions may be revisited in a later release.

A Description of the Proxy Device Protocol

The proxy devices use one or more client drivers, and one server program, in place of the usual drivers. Catalina has two client drivers – one replaces the local SD card plugin when the **PROXY_SD** symbol is defined (e.g. on the command line), and the other replaces up to three HMI drivers when the **PROXY_SCREEN**, **PROXY_MOUSE** or **PROXY_KEYBOARD** symbols are defined (e.g. on the command line). It is possible to mix and match any combination of local and proxy devices. Usually, the server program is compiled to support all devices - the client can then choose to use a real driver to access the local device, or a proxy driver to access a remote device.

The two proxy drivers (i.e. the one for the SD device and the one for all HMI devices) share a single serial connection to the proxy server running on a remote CPU, and coordinate their operation using a Propeller **lock** – this ensures that requests from the two drivers do not conflict even though they are running independently on separate cogs.

There is only one server program, which services all the proxy device requests. It does not need to use a lock – since only one request is sent at a time, it simply services each request as it arrives.

The server is a pure server – it does not ever initiate requests. However, to avoid having to have the client poll the server (e.g. to see if there is a key available) the server does send a signal byte (currently a null byte) to the client continuously whenever it has information waiting to be retrieved – the client does not know from

this whether it is mouse, keyboard or some other data, so when it sees the signal byte it must poll for all possible types of data.

The serial protocol is quite simple, but does employ byte stuffing to avoid accidentally interpreting arbitrary binary data (e.g. SD sector data) as a proxy request or response, and also employs LRC checks on long packets to detect data errors – on some systems the inter-CPU serial communications is a bit noise-prone – possibly due to interaction with the USB Prop Plug on systems that use the Propeller SI/SO pins for serial communications between CPUs. LRC checking is a way to avoid data corruptions on noisy communications channels.

Byte stuffing is done on all (hex) **FF** bytes – they are stuffed to **FF 00**. This is to avoid any possible conflict with the ‘sync’ signal, which is always **FF nn**, where **nn** is the CPU number of the proxy server.

The protocol messages are as follows:

SD_Init – enable (initialize) the SD card

Request: FF nn 01

Response: FF nn 01

SD_Read – read a sector from the SD card

Request: FF nn 02 ss ss ss ss

Response: FF nn 02 <512 bytes> cc

(or) FF nn 00

Note: ss ss ss ss is the sector number
cc is the LRC of the sector data

Note: The short response indicates an error condition.

SD_Write – write a sector to the SD card

Request: FF nn 03 ss ss ss ss <512 bytes> cc

Response: FF nn 03

Note: ss ss ss ss is the sector number
cc is the LRC of the sector data

SD_Bytelo – write a byte to the SD card

Request: FF nn 04 bb

Response: FF nn 04

Note: bb is the byte to write.

SD_StopIO – disable (tristate) the SD card

Request: FF nn 05

Response: FF nn 05

KB_Reset – reset the keyboard (clear any buffered keys)

Request: FF nn 06

Response: FF nn 06

KB_Data – read a character of keyboard data

Request: FF nn 07

Response: FF nn 07 kk

(or) FF nn 00

Note: The short response indicates no keyboard data available.

Note: kk is the keyboard character.

MS_Data – read mouse data

Request: FF nn 08

Response: FF nn 08 xx xx xx xx yy yy yy yy zz zz zz zz bb

(or) FF nn 00

Note: The full response is sent if the server detects any change in mouse data, otherwise the short response is sent.

Note: xx xx xx xx is the abs_x value
yy yy yy yy is the abs_y value
zz zz zz zz is the abs_z value
bb is the button states

TV_Data – write screen data

Request: FF nn 09

Response: FF nn

Request: bb <up to MAX_TEXT bytes> FF nn

Response: FF nn

This protocol will be expanded over time to include new proxy devices, and also to add currently unsupported capabilities to existing proxy devices (such as screen cursor, scroll and color functions).