Catalina

C

Compiler

Reference Manual for the Propeller 2

Table of Contents

What is Catalina? 6
- Status 6
- Features 6
- Catalina is ANSI C compliant 7
- Catalina runs on Windows, Linux and the Propeller 2 7
- Catalina supports multiple Propeller platforms 7
- Catalina supports C programs up to 16 Mb 9
- Catalina is Free! 10
  - But what does all this really mean? 12

Installing Catalina 13
- Overview 13
- Catalina Directory Structure 14

Using Catalina 16
- Using the Catalina Compiler 16
- Catalina Environment Variables 19
- Using lcc directly 21
- Using the Catalina Binder 21
- Using the Payload Loader 23
  - Interactive Mode 27
    - Internal terminal emulation 27
    - External terminal emulation 29
  - A Note about payload's internal interactive mode under Windows 30
  - Catalina YModem Support 31
  - Building the Payload Loader utilities 36
- Quick Build 36

Catalina Support for the Propeller 39
- SPIN/PASM Assembler Support 39
- Clock Configuration support 39
- Memory Management Support 40
  - The hub_malloc functions 41
  - The alloca function 42
  - Specifying the heap size 43
- Floating Point Support 44
- HMI Support 45

Keyboard functions..... 46
Mouse functions..... 47
Screen functions..... 48
Utility functions..... 50
VGA and USB Support..... 51
VGA Fonts..... 53
Multi-Processing Support..... 53
Multi-processing and plugins..... 53
Multi-Cog Support..... 54
Multi-Threading Support..... 55
Fundamental Thread Functions..... 55
Additional Thread Utility Functions..... 58
Multi-Model Support..... 59
Producing an array blob:..... 60
Multi-Models and Interrupts..... 62
Multi-Models and Multi-threading..... 62
Multi-Memory Models and Overlays..... 63
Multi-processing, Locks and Memory Management..... 64
Thread-safe Memory Management..... 64
Memory Fragmentation Management..... 65
Thread, Memory and Service Locks..... 66
Posix threads (pthreads) support..... 67
Posix pthread functions supported..... 67
Posix pthread functions not supported..... 70
Non-Posix pthread functions added..... 70
Interrupt Support..... 70
Fundamental Interrupt Functions..... 71
Interrupt Examples..... 73
Random Number Support..... 74
Pseudo Random Numbers..... 74
True Random Numbers..... 74
Random Number Functions..... 74
Plugin Support..... 75
Cog functions..... 75
propeller2.h, propeller.h and Special Register Access..... 83
Registry, Plugin and Service functions..... 84
Debugger Support..... 89
SD Card Support..... 89

Real-Time Clock Support....89
Flash Support....91
PSRAM and HyperRAM/HyperFlash Support....91
Using PSRAM and HyperRAM/HyperFlash as additional storage....92
Using PSRAM and HyperRAM as XMM RAM....95
File System Support....96
Serial Device Support....100
The 2 port Serial library (libserial2)....101
The Multi Port Serial (aka 8 port Serial) library (libserial8)....103
Cache Support....107
LUT Execution Support....108
LUT_BEGIN(OFFS, NAME, SIGN)....109
LUT_END....109
LUT_CALL(NAME)....109
Using the LUT Execution Macros....110
Lua Support....111
Embedding Lua scripting in a C program....112
Catalina Targets....114
Catalina Propeller 2 Targets....114
Default Target Configuration Options....115
Catalina Hub Resource Usage....118
LMM Support....118
CMM Support....119
NMM Support....120
XMM Support....121
Specifying the Memory Model....121
Catalina Cog Usage....123
Supporting multiple Propeller platforms....124
Target Packages....125
The standard target package (target\p2)....125
Using PASM with Catalina....126
Inline PASM....127
The PASM function....127
The _PASM macro....127
The _PSTR macro....130
Precautions when using LMM PASM with the Catalina Optimizer....133
Load the PASM program at initialization time....133
Convert the PASM program into a Catalina plugin....134

Load a compiled PASM program into a cog....134
Write a PASM function that can be called from C....135
Precautions when using PASM with the Catalina Optimizer....135
Parallelizer Support....136
Customizing Catalina....137
Customized Platforms....137
Building Catalina....138
Catalina Technical Notes....139
A Note about Binding and Library Management....139
A Note about the Catalina Libraries....141
A Note about C Program Startup & Memory Management....142
A Note about Catalina Code Sizes....145
A Note about Catalina symbols vs C symbols....149
A Note about the Catalina Loader Protocol....151
A Note about Self-hosted Catalina....152
A Note about DOS 8.3 File names used by Self-hosted Catalina....154
Catalina Development....160
Reporting Bugs....160
If you want to help develop Catalina....160
Okay, but why is it called “Catalina”?....160
Acknowledgments....161
Catalina Internals....163
A Description of the LMM Kernel....163
A Description of the Catalina LMM Virtual Machine....164
Registers....164
Primitives....166
Unsupported PASM....170
A Description of the Catalina NMM Virtual Machine....171
Registers....171
Primitives....172
Unsupported PASM....174
A Description of the Catalina Addressing Modes....174
Catalina Calling Conventions....175

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 is an 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 2**. Some of the libraries, and several of the supported memory models and plugins are only supported on the Propeller 1. For information specific to the Propeller 1, see the ***Catalina Compiler Reference Manual for the Propeller 1***.

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 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 512 kb - up to 16 Mb;
- 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 required[1].

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 the Propeller 2

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 has been built from source to run on OSX, although this is not supported "out of the box".

Catalina can also run on the Propeller 2 itself using the Catalyst operating system. The Propeller 2 must have 32Mb of suitable PSRAM. Currently supported are the P2 EDGE module equipped with PSRAM (i.e. the P2-EC32MB), or a P2 EDGE or P2 EVAL board equipped with the Parallax HyperRAM add-on board. See the section in this manual titled **A Note about Self-hosted Catalina** for more technical detail, and the **Catalyst Reference Manual** for usage details.

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).

---

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

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).

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 2, Catalina compiles C programs into **LMM** (i.e. Large Memory Mode, which for historical reasons is also known as **TINY** mode), **CMM** (i.e. Compact Memory Mode, also known as **COMPACT** mode) or **NMM** (i.e. Native Memory Mode, also known as **NATIVE** mode) or **XMM** (i.e. eXternal Memory Mode, also known for historical reasons 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 currently provides just one target package for the Propeller 2, in a specific sub-directory[2]:

target\p2 This is the default Catalina target package for the Propeller 2. It supports multiple Propeller platforms, all memory models, all load options, and all plugins[3].

This default target package is flexible enough to accommodate all the possible hardware configurations of all the supported Propeller 2 platforms. Other target packages may be added in subsequent releases.

The base **Propeller 2** platforms currently supported are:

- The Parallax **P2 EVAL** board
- The Parallax **P2_EDGE** board
- The **P2D2** board
- The **P2_CUSTOM** boards (by default this is configured the same as the Parallax **P2_EVAL** board)

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.

---

[2] You may also see sub-directories with a **\p1** suffix, such as **target\p1** – these are specific to the Propeller 1. Refer to the Catalina C Compiler Manual for the Propeller 1 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.

There is also a default platform, used if no other platform is specified on the command line (more on how to do this later).

The **P2_EVAL** platform comes pre-configured to be suitable for either RevA or RevB of the Parallax P2 EVAL (64000-ES) board.

The **P2_EDGE** platform offers support for the version with or without PSRAM. However, XMM programs are only supported on the version *with* PSRAM.

Both the **P2_EVAL** and **P2_EDGE** boards can use the Parallax HyperFlash/HyperRAM add-on board as XMM.

The **P2_CUSTOM** platform is intended to be 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\p2*) and the **P2_CUSTOM** platform are in use, although there is currently no difference between this platform and the **P2_EVAL** platform.

Catalina supports C programs up to 16 Mb

Catalina supports both **NATIVE** (NMM) and **COMPACT** (CMM) programs up to 512 kb on *any* Propeller 2 platform. It also supports **TINY** (LMM) programs on any Propeller 2 platform, but this model is a legacy of the Propeller 1, and is not used much on the Propeller 2 except as the basis for **SMALL** and **LARGE** (XMM) programs.

The advantage of the **NATIVE** model is that it allows the Propeller to execute at full speed using the Propeller Hub Execution mode.

The advantage of the **COMPACT** model is that it typically *halves* code sizes when compared with **NATIVE** models.

However, no matter how efficient the compiler, sometimes a program is just *too large* to fit in 512 kb – so Catalina also provides **External Memory Model** (XMM) support for programs larger than 512 kb on suitable platforms. XMM support allows Catalina to support program sizes up to 16 Mb on Propellers equipped with suitable hardware, such as the PSRAM on some **P2_EDGE** boards, or the Parallax HyperFlash/HyperRAM add-on board.

.

Models other than **NATIVE** are not executed directly on a “bare metal” Propeller – instead, a *kernel* is first loaded into one or more of the Propeller cogs and these cogs can then execute LMM or CMM programs. However, LMM and XMM programs are not “interpreted” in quite the same way as CMM or SPIN programs – the binary opcodes are true Propeller opcodes – the main difference between an LMM or XMM program and an NMM program is that for NMM mode programs the program code executed directly from Hub RAM using the Propeller 2's Hub execution mode. For LMM programs the opcodes are fetched from Hub or External RAM first, and then executed in Cog RAM. This means LMM and XMM programs are slower than Native

programs – but XMM programs can be significantly *larger* than Native mode programs (which are limited to 512 kb) if suitable external memory is available.

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

Catalina provides a Standard Target Package which includes support for NMM, CMM and LMM programs on all platforms[4]. XMM is currently supported only on platforms that have suitable PSRAM, such as the **P2_EDGE**, or via the Parallax HyperFlash/HyperRAM add-on board, which can be added to the **P2_EVAL** or **P2_EDGE** boards.

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 (GLPGL), 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:

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

[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**[5] 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.

[5] 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 2, the various **p1** sub-directories will not be discussed in detail. Refer to the **Catalina C Reference Manual for the Propeller 1** 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 customized version of the **Geany** Integrated Development Environments (IDE), 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 the Geany IDE 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 Assembler (**p2asm**), 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 -p2 hello_world.c -lc
```

By default, Catalina compiles each C file specified on the command line to a PASM file, then includes additional files for any required library functions, combines the

results into a single file and then invokes **p2asm** 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 **-p2** option to the **catalina** command above specifies that we want to compile a program for the Propeller 2, not the Propeller 1.

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

-? or **-h**	print this help (and exit)
-b	generate a binary output file (this is the default)
-B baud	baud rate to use for serial interfaces (P2 only)
-c	compile only (do not bind)
-d	output diagnostic messages
-C symbol	define Catalina symbol (e.g. **-C HYDRA**)
-D symbol	define symbol (e.g. **-D printf=tiny_printf**)
-e	generate an eeprom output file
-g[level]	generate debugging information (default level = 1)[6]
-H addr	address of top of heap
-I path	include file path (e.g. C:\Program Files (x86)\Catalina\include)
-l lib	search library lib when binding
-k	kill (suppress) the output of compilation statistics
-L pat	path to libraries (e.g. C:\Program Files (x86)\Catalina\lib)
-M size	maximum memory size (use with -x)
-o name	name of output file (default is first file name)
-O[level]	optimize code (default level = 1)[7]
-p ver	Propeller Hardware Version (ver = 1 or 2)
-P addr	address for Read-Write segments
-q	Enable Quick Build (see the section on **Quick Build** in this document)
-R addr	address for Read-Only segments

[6] 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

[7] 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.

`-R size`	size of Read/Write segments
`-S`	compile to assembly code (do not bind)
`-t name`	name of dedicated target to use
`-T path`	path to target files (e.g. C:\Program Files (x86)\Catalina\target)
`-U symbol`	undefine symbol (e.g. **`-U DEFAULT`**)
`-v`	verbose (output information messages)
`-v -v`	very verbose (more information messages)
`-W option`	option to pass directly to **lcc**
`-x layout`	use specified segment layout (layout = 0 .. 6, 8 .. 11)
`-y`	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 -p2 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 Propeller 2 Evaluation platform and the high-resolution VGA driver from the target package, you might use a command like:

```
catalina -p2 hello_world.c -lc -C P2_EVAL -C HIRES_VGA
```

For more details on support for particular Propeller 2 platforms check if there is a file called ***platform*_README.TXT** in the *target\p2* directory. For example:

P2_EVAL_README.TXT

P2_EDGE_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 -p2 "C:\Program Files (x86)\Catalina\demos\hello_world.c" -lc -y
```

Catalina Environment Variables

The Catalina Compiler can also use the following environment variables:

CATALINA_DEFINE	a list of symbols to define before the compilation
CATALINA_INCLUDE	a list of paths to search for include files
CATALINA_LIBRARY	the directory where libraries are located
CATALINA_TARGET	the directory where target files are located
CATALINA_TEMPDIR	the directory **lcc** will use for temporary files
CATALINA_LCCOPT	any options to be passed directly to **lcc**
LCCDIR	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 **P2_EVAL**

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. When the **-p2** option is also specified, the compiler will automatically add **\p2\lmm** to the library path for **TINY** or **XMM SMALL** programs, **\p2\cmm** for **COMPACT** programs, and **\p2\nmm** for **NATIVE** programs and **\p2\xmm** for **XMM 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. When compiling for the Propeller 2 (i.e. including the **-p2** option), 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\p2\lmm	default sub-directory for library files when using TINY mode or XMM SMALL mode.
lib\p2\cmm	default sub-directory for library files when using COMPACT mode
lib\p2\xmm	default sub-directory for library files when using XMM LARGE mode
lib\p2\nmm	default sub-directory for library files when using NATIVE mode
include	default sub-directory for include files
target\p2	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 that it is possible to specify binder options to **lcc** by prefixing them with **–Wl** – i.e. the **lcc** option **–Wl–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 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 PASM file **file.s** with the math library and then generate an eeprom image containing the result named **test**, use the following command:

```
catbind -p2 file.s -lm -e -o test
```

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

```
catbind -p2 -i -e *.s -o catalina.index
```

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:

-? or **-h**	print this helpful message (and exit)
-a	no assembly (output bound source files only)
-d	output diagnostic messages (-d -d for even more messages)
-C symbol	#define 'symbol' before assembling the code
-e	generate export list from input files
-f	force (continue even if errors occur)
-i	generate import list from input files
-k	kill (suppress) the output of compilation statistics
-L path	path to libraries (default is 'C:\Program Files (x86)\Catalina\lib\')
-l name	search library named 'libname' when binding
-M size	memory size to use (used with -x, default is 16M)
-o name	output results (generate, bind or assemble) to file 'name'
-O[level]	optimize code (default level = 1)[8]
-p ver	Propeller Hardware Version (ver = 1 or 2)
-q	Enable Quick Build (see the section on **Quick Build** in this document)
-R size	size of Read/Write segments
-T path	target file path (default 'C:\Program Files (x86)\Catalina\target')
-t name	use target 'name'
-u	untidy mode – do not delete intermediate files
-U symbol	do not #define 'symbol' before assembling the code
-v	verbose (output information messages)
-v -v	very verbose (more information messages)
-w opt	pass option 'opt' to the assembler (e.g. -w-l, -w-b, -w-e)
-x layout	use specified segment layout (layout = 0 .. 6, 8 .. 11)
-z ch	specify separator char for path names (default is '\')

[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.

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**.

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 includes a built-in terminal emulator.
- It can auto-detect both Propeller 1 and Propeller 2 chips.
- It can be used to load programs to either RAM or FLASH RAM

At its simplest, payload is quite trivial to use. For example, to load **program.bin** on the Propeller 2, 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
```

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 2 will respond to a program

being loaded via a serial port on pins 62 and 63. However, this built-in loader can only be used to load programs into Hub RAM – it knows nothing about any external XMM RAM 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.bin*, 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.bin*) 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.

On the Propeller 2, a payload can be used to load programs into FLASH RAM. A script to load a program to FLASH is provided, called **flash_payload**. A command to do this might look as follows:

```
flash_payload my_program -o2 -b230400
```

See the section on **Flash Support** for more details on programming FLASH RAM.

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 to enable translation of CR to LF on output). Line modes can be combined.
`-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 all lines with both a CR and an LF) to be used in the built-in interactive terminal (which assumes Linux style, which terminates all lines with an LF only).

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

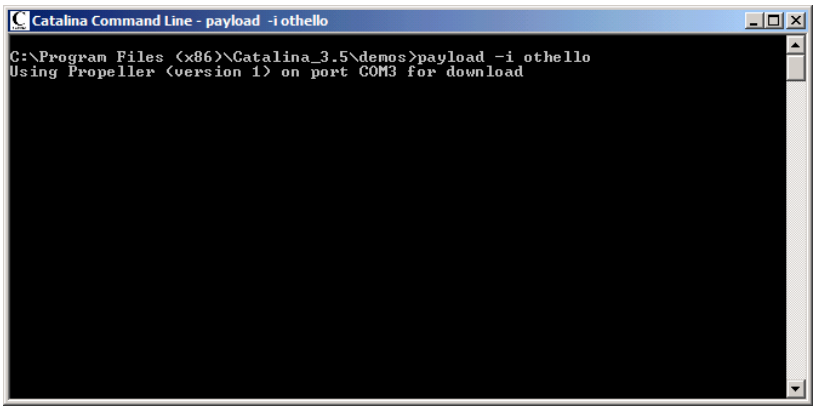
In addition to just loading programs, payload can be used to interact with the Propeller if the loaded program is compiled to use a serial interface (such as **TTY** or **SIMPLE**). Payload offers a simple internal terminal emulator which is adequate for many purposes, and can also invoke an external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

Internal terminal emulation

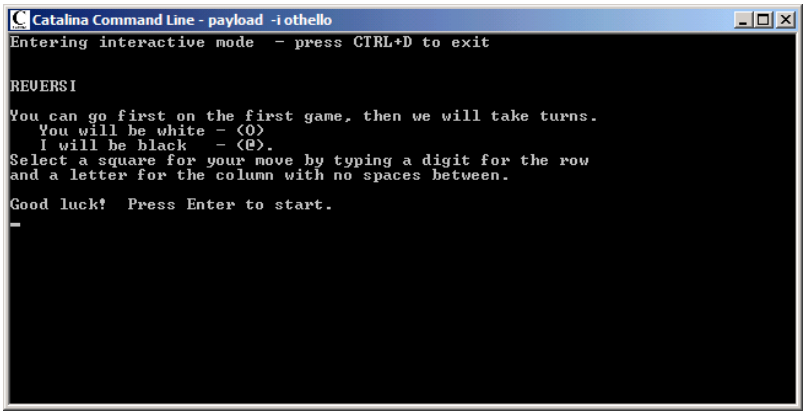
The internal terminal emulator is invoked by adding **-i** to the payload command. For example:

```
catalina -p2 othello.c -lc -C TTY
payload -b230400 othello -i
```

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



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



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 accepts a terminal name as 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 230400

vt102 /dev/ttyUSB1 230400
```

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** (n 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**).

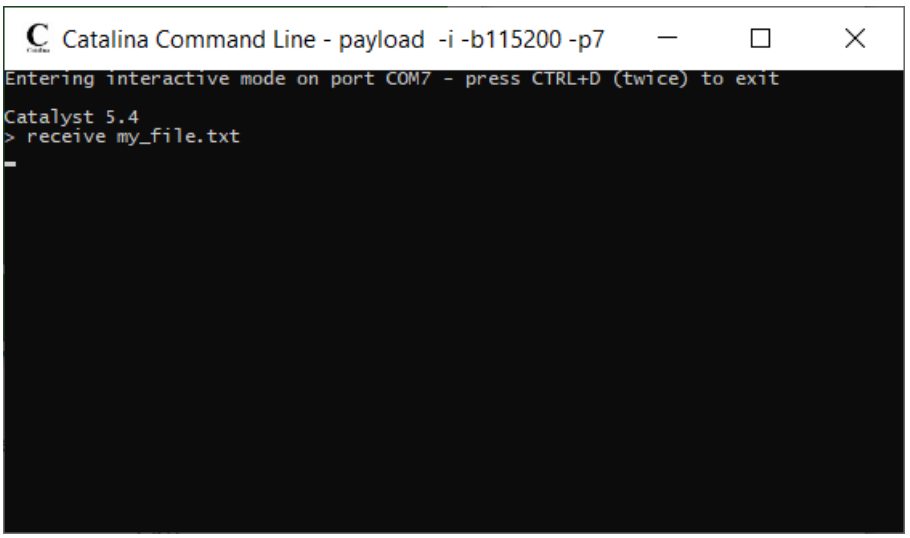
When using a terminal emulator that has YModem support, you typically execute the **send** or **receive** on the Propeller first, and then initiate the YModem transfer in the terminal emulator. For example, with payload, and assuming you have a Propeller 2 connected to COM port N, you would typically start payload as follows:

```
payload -i -pN
```

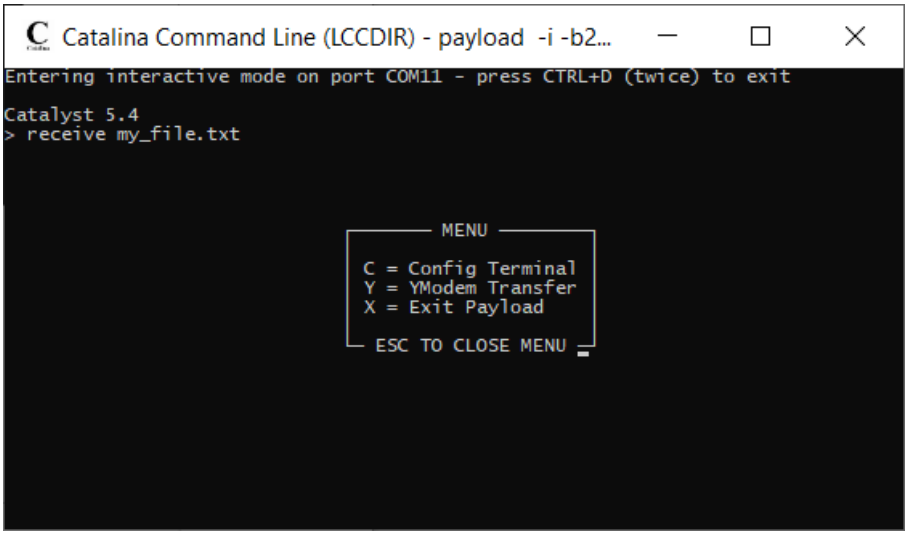

To receive a command, you would enter a command like the following in payload or on your Propeller keyboard:

```
receive my_file.txt
```

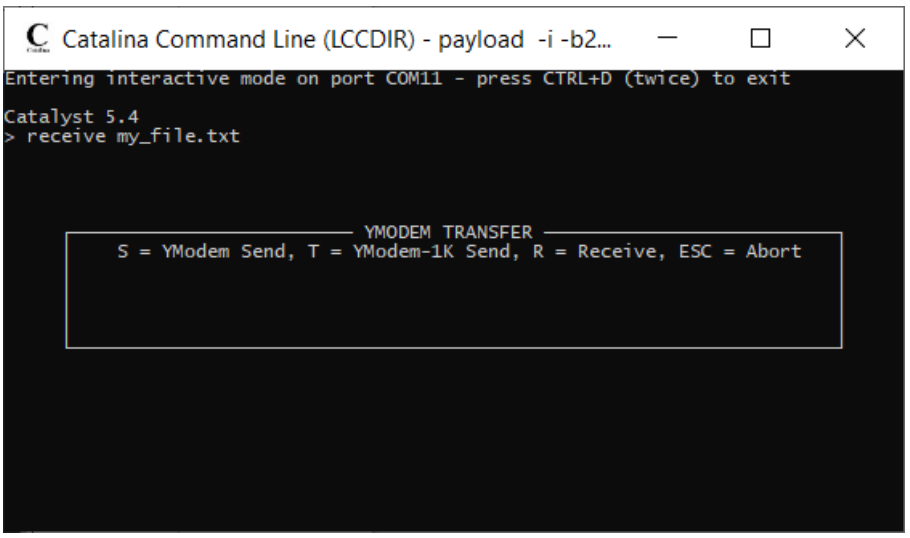
We will assume you are using a serial HMI, so you would then see a screen similar to the following:



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:

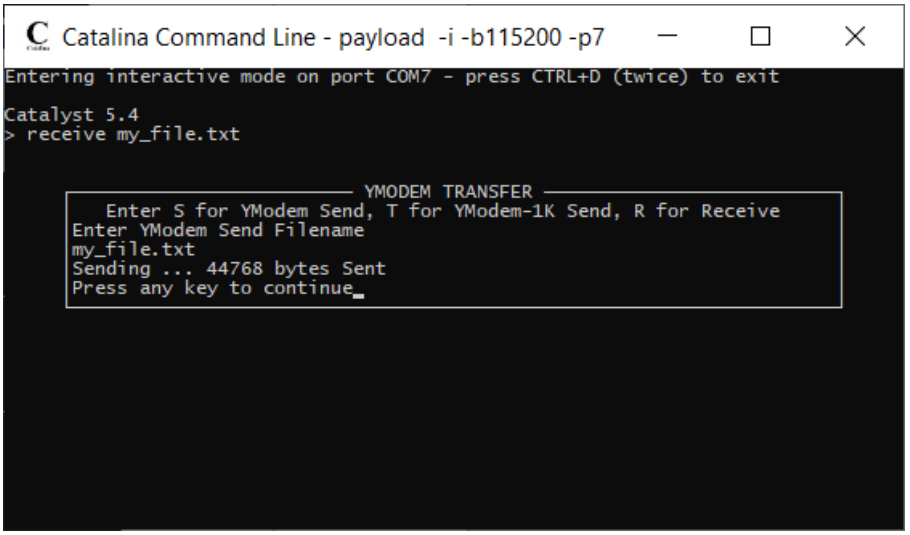


Then press **Y** and the YModem Transfer dialog will appear, as follows:



Since we are using a Propeller 2, we can use YModem-1K (i.e. 1024 byte block) mode and so press **T** to initiate a YModem-1K 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:



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 2:

```
payload -i -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 the following utilities if they are supported for the platform:

SRAM.bin

For historical reasons, the **SRAM.bin** file will also be copied to **XMM.bin**.

This utility allows payload to be used to load **SMALL** or **LARGE** XMM programs into External RAM on platforms that support it (e.g. using the Parallax HyperFlash/HyperRAM add-on board on the **P2_EVAL** or **P2_EDGE**, or the PSRAM on the **P2_EDGE**).

For example (assuming we built the utilities for the **P2_EDGE** with a cache size of 64K):

```
catalina -p2 othello.c -lci -C P2_EDGE -C SMALL -C CACHED_64K
payload SRAM othello
```

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

Quick Build

On the Propeller 2, Catalina normally builds **NATIVE**, **TINY** and **COMPACT** programs *monolithically* - i.e. there is only one file generated, which contains both the target (i.e. the run-time environment) and the compiled program code. No special loader is required for such programs - they can simply be loaded into Hub RAM (either serially or from SD card) and executed in exactly the same way as any other Propeller program produced by any other assembler or compiler.

On the other hand, XMM programs (**SMALL** and **LARGE**) always generate *two* files - one contains the target plus the special loader required to load the program into XMM RAM, and the other contains the compiled program code. Catalina then combines these two files into a single binary, which must be loaded using **Catalyst**, or with **payload** using special utilities. Normally, the target file is discarded and rebuilt on each compilation.

Enabling Quick Build allows the same target file to be used for multiple compilations, significantly reducing compile times when Catalina is used repeatedly to compile programs for the same target, such as when debugging a C program for a specific target, or using the self-hosted version of Catalina.

Quick Build is enabled by adding the **-q** option to the catalina command, or defining the Catalina symbol **QUICKBUILD**. This does several things, depending on the memory model in use:

1. It causes Catalina to check if a target file already exists, and (if so) use it instead of rebuilding it. Note that the compiler only checks the *name* of the file

- it assumes the file has been compiled with the correct options for the current compilation (more on this below).

2. It stops Catalina deleting the target file after the current compilation completes, so that it may be used again in subsequent compilations. Note that a target file may also exist after the **-u** (untidy) command line option is used.

3. For **NATIVE**, **COMPACT** and **TINY** programs, it causes Catalina to separate the compilation into two parts - the target and the program, which are then combined into a single binary (as is already the case for XMM **SMALL** and **LARGE** programs). The target file is left intact. This makes **NATIVE**, **COMPACT** and **TINY** executables larger when Quick Build is enabled (more on this below).

The target file will be named according to both the memory model in use and the target in use. The default target is **def**, but a different target can be specified via the Catalina **-t** option. This means that the following would be common target files generated when using Quick Build:

`cmmdef.bin`	the default target name for **CMM** (i.e. **COMPACT)** programs
`lmmdef.bin`	the default target name for **LMM** (i.e. **TINY)** programs
`nmmdef.bin`	the default target name for **NMM** (i.e. **NATIVE)** programs
`xmmdef.bin`	the default target name for **XMM** (i.e. **SMALL** or **LARGE**) programs

The target file must be in the current directory when the compilation is performed, but target files can be copied and saved to other directories to preserve them if required. If a target file is accidentally deleted, it will be regenerated on the next compilation as required. However, if the Catalina options that affect the target file are changed, the existing target file should either be manually deleted, or else the next compilation should NOT enable Quick Build, to force Catalina to rebuild it as required.

Here are the Catalina options that will affect the target:

- the platform selected (e.g. by defining **C3**, **P2_EDGE** etc)
- the plugins selected (e.g. by defining **CLOCK**, **RTC**, **VGA**, **TTY** etc)
- the clock selected (e.g. via **-f**, **-e** and **-E**, or defining **MHZ_200** etc)
- the memory model selected (via **-x** or defining **TINY**, **NATIVE**, **COMPACT**, **SMALL** or **LARGE**)
- the floating point option selected (via **-lm**, **-lma**, **-lmb** or **-lmc**)
- whether **NO_PLUGINS**, **NO_ARGS** or **NO_ENV** is defined

- whether multi-threading is selected (via **-lthreads**)
- whether an SD Card plugin is selected (via **-lcx**, **-lcix** or defining **SD**)
- whether debugging is selected (via **-g** or **-g3**)
- using custom memory sizes and/or addresses (via **-M**, **-P** or **-R**)
- specifying a specific address for the heap top (via **-H**)

When Quick Build is enabled for **NATIVE**, **TINY** or **COMPACT** programs, the program code must be compiled to run at a specific address, because the details of the target are not known when the program code is compiled (and vice versa). A target can potentially be up to 64kb - this allows for a target that does a complete load of all 16 cogs, including loading both Cog RAM and LUT RAM, which can be 2k bytes each. This means that when compiled using Quick Build, the program will always be 64k bytes larger than usual. In some cases the resulting binary may no longer fit in Hub RAM, or may not work correctly even if it does fit. In such cases,the program may be able to be compiled as an XMM program. If not, then Quick Build cannot be used. In any case, it is recommended that Quick Build be used only during development, and the final program be compiled without enabling Quick Build, to maximize the Hub RAM available to the program.

If in doubt, simply do not enable Quick Build on the first compilation, or manually delete any existing target files before the first compilation. After that, Quick Build can be enabled for subsequent compilations unless the compile options that affect the target need to be changed.

Catalina Support for the Propeller

SPIN/PASM Assembler Support

On the Propeller 2, Catalina uses **p2asm** as its default SPIN/PASM assembler. Sources for this are included. The main changes to p2asm required for Catalina are two definitions in the file *symsubs.h*:

```
#define MAX_SYMBOLS       50000
#define MAX_SYMBOL_LEN    65
```

However, to make **p2asm** behave more like **spinnaker** (which is the assembler used on the P1, p2asm is invoked via a script called **p2_asm.bat** (or just **p2_asm** on **Linux**).

By default, Catalina uses the **-v33** option to p2asm when compiling programs, which is suitable for RevB of the P2_EVAL board, or a non-evaluation Propeller 2 board. If it ever becomes necessary to disable this option (e.g. when compiling programs for a RevA **P2_EVAL** board, or one of the FPGA implementations of the Propeller 2) then this option can simply be removed from the **p2_asm** script.

Clock Configuration support

Each supported Propeller 2 platform can specify default clock parameters in the various include files specified in the *platform.inc* file, but you can also request a specific clock frequency, or specify arbitrary clock configuration parameters, on the Catalina command line.

Specifying the clock frequency is done using the **-f**, **-F** and **-E** command line options:

-f requested frequency for which to calculate clock parameters

-F xtal (XI) frequency to use in frequency calculation (default 20Mhz)

-E error limit for frequency calculation (default 100khz)

In most cases, you just use the **-f** option by itself. Note that **m** and **k** suffix chars (case insensitive) are supported by these options, and mean to multiply the value specified by 1,000,000 or 1,000 respectively. This means you can say things like:

```
-f 260Mhz
```

or

```
-f 123456kHz
```

for example:

```
catalina hello_world.c -p2 -lc -f300Mhz
```

Specifying a frequency via **-f** causes Catalina to calculate the clock parameters required to achieve it, and defines and uses three Catalina symbols if it is possible to achieve the specified frequency within the error limit. These symbols are:

_CLOCK_XDIV	XI divider (1..64)
_CLOCK_MULT	XI multiplier (1..1024)

_CLOCK_DIVP VCO divider (1, or even numbers from 2 to 30)

These symbols can also be defined manually, but they will only be used if all three are defined.

The following clock parameters can also be defined:

_CLOCK_XTAL XI frequency
_CLOCK_OSC 0=OFF, 1=OSC, 2=15pF, 3=30pF
_CLOCK_SEL 0=rcfast, 1=rcslow, 2=XI, 3=PLL
_CLOCK_PLL 0=PLL off, 1=PLL on

To manually specify the value for clock configuration parameters, use complex symbol definitions, which means you need to enclose them in double quotes – e.g:

```
-C "_CLOCK_SEL=2"
```

or

```
-C "_CLOCK_XTAL=25000000"
```

or

```
-C "_CLOCK_XDIV=1" -C "_CLOCK_MULT=9" -C "_CLOCK_DIVP=1"
```

Note that the following two command-line options have the same effect:

```
-C "_CLOCK_XTAL=25000000"
```

and

```
-F 25Mhz
```

but note that the "25Mhz" notation is not supported when using the **-C** option. This means you **CANNOT** say:

```
-C "_CLOCK_XTAL=25Mhz"    <-- WRONG!
```

Note that defining the Catalina symbols **MHZ_260**, **MHZ_220** or **MHZ_200** has the same effect as the options **-f 260MHz,** -**f 220MHz** or **-f200Mhz** respectively. These allow common frequencies to be specified using the **CATALINA_DEFINE** environment variable. For example:

```
set CATALINA_DEFINE = P2_EDGE MHZ_200
catalina hello_world.c -p2 -lc
```

would achieve the same as:

```
catalina hello_world.c -p2 -lc -C P2_EDGE -f200Mhz
```

If conflicting clock parameters are specified, the last one on the command line will be used. If no clock parameters are specified, or the requested frequency cannot be achieved, the default clock parameters defined for the platform in *platform.inc* will be used.

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 primarily intended to be used 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_C**. All the math functions supported by Float32Full are implemented using the CORDIC functions of the Propeller 2 in one cog, with software emulation of other required functions not implemented in Float32Full (e.g. sinh, cosh tanh). This is the best solution for Propeller 2 programs that can spare a cog. Requires that the **mc** library be used (i.e. use command line option **–lmc**).
- **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 **mb** 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 **ma** 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 **m** 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, but unless you need exact compatibility with the floating point options offered on the Propeller 1 (which has only **-lm, -lma** and **-lmb**) on the Propeller 2 there is really no reason to use anything other than **-lmc** if you have a spare cog available, or **-lm** if not. The CORDIC functions are both faster and more accurate than the alternatives.

HMI Support

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

VGA support:

- Resolution:
  - High-resolution (**HIRES_VGA** or **VGA_1024**) - 128 x 48 chars;
  - Medium resolution (**VGA_800**) - 100 x 40 chars;
  - Low resolution (**LORES_VGA** or **VGA**) - 80 x 30 chars;
- Color depth:
  - 1 bit color (**COLOR_1** or **MONO**);
  - 4 bit color (**COLOR_4**)
  - 8 bit color (**COLOR_8**)
  - 24 bit color (**COLOR_24**)

USB Keyboard

USB Mouse

Terminal emulator support:

- **TTY** serial terminal emulator;
- **SIMPLE** serial terminal emulator

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. As far as possible, Catalina attempts to detect if an unsupported HMI configuration is specified.

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 CR 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.

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 serial 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 serial HMI drivers.

Mouse functions

```
int m_present();
```

This function returns one if a mouse is detected, or zero otherwise. Not supported (always returns zero) on serial 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 serial 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 the **ci** or **cix** libraries.

```
int t_hex(unsigned curs, unsigned val);
```
[9]

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);
```
[10]

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. The VGA drivers implement two independent cursors. 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

[9] 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.

[10] Ditto.

1 – cursor is an underscore

bit 3: 0 – cursor wraps at end of screen

1 – screen scrolls end of screen

Note that cursor 0 is always invisible – the visibility bits only apply to cursor 1.

```
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 background and foreground color. For **COLOR_4** mode each color is 3 bits (0 to 7) of ANSI background color and 4 bits (0 to 15) of ANSI foreground color. For **COLOR_8** mode each color is 8 bits of ANSI color. (For **COLOR_24** mode, use the ***t_color_fg()*** and ***t_color_bg()*** functions described below). Each character cell can have its colors set independently of the others. The new colors are applied to any new characters output to the screen - to apply the specified color to the whole screen, you can clear the screen.

The two colors are specified as a 16 bit number (as bg * 256 + fg).

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

This function sets the screen foreground color. For **COLOR_4** mode, the parameter is 4 bits of ANSI fg color. For **COLOR_8** mode it is an 8 bit ANSI color, and for **COLOR_24** mode it is 24 bits of color - 8 bits each of RGB. 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. Note that the curs parameter is ignored - the new colors apply to both cursors.

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

This function sets the screen background color. For **COLOR_4** mode, the parameter is 4 bits of ANSI fg color. For **COLOR_8** mode it is an 8 bit ANSI color, and for **COLOR_24** mode it is 24 bits of color - 8 bits each of RGB. 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. Note that the curs parameter is ignored - the new colors apply to both cursors.

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 the **ci** or **cix** library.

%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);
```

VGA and USB Support

The Catalina VGA HMI option includes support for a VGA driver of configurable resolution and color depth, and either one or two USB drivers, which can be used fora keyboard and/or a mouse. Do not connect two keyboards or two mice as they will conflict, and do not connect the USB devices via a USB hub – they must be plugged in directly to be recognized correctly by the USB drivers.

The following Catalina symbols can be defined on the command line (using the -C option) to modify the behavior of the VGA plugin:

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.
HIRES_VGA	same as **VGA_1024**
LORES_VGA	same as **VGA_640**
VGA	same as **VGA_640**
COLOR _1	color depth is 1 bit.
COLOR _4	color depth is 4 bits.
COLOR _8	color depth is 8 bits.
COLOR _24	color depth is 24 bits.
MONO	same as **COLOR_1**.
NO_KEYBOARD	do not load the keyboard code and one USB driver
NO_MOUSE	do not load the mouse code and one USB driver
P2_REV_A	use instructions supported by the P2 Rev A chip(required for Rev A chips).

MHZ_220	use a clock frequency of 220 Mhz (required for High resolution VGA mode).
MHZ_260	use a clock frequency of 260 Mhz (required for High resolution VGA using 4 bit or 8 bit color).

Note that the high resolution VGA options (i.e. **HIRES_VGA** or **VGA_1024**) requires a minimum clock speed of 200 MHz for the **COLOR_1** (or **MONO**) or **COLOR_24** options, and a minimum of 260 Mhz for the **COLOR_4** and **COLOR_8** options. The clock speed can be set on the command line using the **MHZ_220** or **MHZ_260** options. For example:

```
catalina -p2 -lci test_vga.c -C P2_EVAL -C VGA_800 -C COLOR_24
catalina -p2 -lci test_terminal.c -C P2_EVAL -C VGA_1024 -C COLOR_8 -C
MHZ_260
```

Except as noted below, the ASCII keys on the keyboard will return the appropriate ASCII codes. The non-ASCII keys will return key codes compatible with the P1 HMI keyboard plugins (which themselves were based on the original Parallax keyboard codes):

HEX	MEANING	NOTE
$08	Backspace	Original Parallax drivers returned $C8
$09	Tab	
$0a	Enter	Returns $0d if NO_CR_TO_LF specified
$1B	Esc	Original Parallax drivers returned $CB
$C0	Left Arrow	
$C1	Right Arrow	
$C2	Up Arrow	
$C3	Down Arrow	
$C4	Home	
$C5	End	
$C6	Page Up	
$C7	Page Down	
$C9	Delete	
$CA	Insert	
$D0..$DB	F1..F12	
$DC	Print Screen	
$E0..$E9	Keypad 0..9	
$EA	Keypad .	

$EB	Keypad Enter
$EC	Keypad +
$ED	Keypad -
$EE	Keypad *
$EF	Keypad /

The pins used by the VGA and USB drivers are configured using the following definitions in the various include files specified in the file *platform.inc* in the *target\p2* subdirectory, which should be modified in the section of the file for the platform you are using:

_VGA_BASE_PIN = 32

_USB_BASE_PIN = 40

The HMI plugin may support other options that can be configured either in this file, or via command-line options – refer to the file *Catalna_HMI_Plugin_VGA.spin2* for more details.

VGA Fonts

Unlike the Propeller 1, the Propeller 2 has no built-in font data. This means font data must be loaded by the VGA HMI plugin, which consumes some Hub RAM space. However, this means the fonts used are configurable. The default font is a public domain font known as unscii-16, but other fonts can be used. Details on how to create suitable font files and configure Catalina to use them are given in the file **README_P2_Fonts.txt** in the *source\catalina* subdirectory

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 **pthreads** 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 *hreads.*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 *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

On the Propeller 2, Catalina allows the Propeller to simultaneously execute LMM, CMM, NMM 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 or on SD Card, and only loaded into Hub RAM when they are to be executed. 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), COMPACT (CMM) or NATIVE (NMM) program, with all the usual command-line options, plus the following additional options:

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


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), NATIVE (NMM), 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.

To run MULTIPLE secondary programs 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 on the Propeller 2 is done using an updated version of the **spinc** utility, using new command-line options (**-p**, **-B**). For example:

```
spinc -p2 -B2 -c secondary_1.bin >blob_1.inc
spinc -p2 -B2 -c secondary_2.bin >blob_2.inc
  (etc)
```

The **-B** option specifies that a blob is to be created from one of the objects in the binary. The parameter to the **-B** option is not currently used on the Propeller 2, but it is recommended it always be specified as 2 for compatibility with the Propeller 1.

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.bin*, 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, CMM or NMM 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 -p 2 -B 2 -s 200 -n my_blob secondary.bin >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      0xXXXX // size of the blob in bytes
#define <<name>>_RUNTIME_SIZE   0xXXXX // runtime size in bytes
#define <<name>>_CODE_ADDRESS   0xXXXX // 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 Interrupts

The Multi-Model support works with programs that use interrupts. Both the primary or the secondary programs can be compiled to use interrupts. XMM programs do not support interrupts, but an XMM primary program can start non-XMM secondary programs that do.

See the file *run_test_interrupts_1.c* in the *demos\multimodel* directory for an example.

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

_threaded_cogstart_NMM_cog() - start a blob as a threaded NMM 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.bin >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.bin
   >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-Memory Models and Overlays

Catalina allows programs to load overlays from files on an SD Card. 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.bin -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.bin -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 mult-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_mutextattr_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

Interrupt Support

Interrupts are new on the Propeller 2. Catalina fully supports the Propeller 2's interrupt capabilities, allowing C functions to serve as interrupt service functions.

Interrupts are supported in C programs that use the NATIVE, COMPACT or TINY memory models. They are not supported in programs that use the XMM (SMALL or LARGE) memory models. However, it is possible to have an XMM C program

execute in parallel with a C program compiled in another mode that uses interrupts, and the two programs can share plugins and Hub RAM. See the section titled **Multi-model Support**.

To use a C function as an interrupt service routine, the function must take no arguments, and it must return - but it cannot return a value.

Here is the typedef that defines such a function:

```
typedef void (* _interrupt)(void);
```

This typedef is used in the **_set_int_x()** function, which is used to specify which interrupt the function will service. This function is defined in the header file *catalina_interrupts.h*. You must specify the type of interrupt, the address of the interrupt function, and some stack space for the interrupt function to use. There is a separate function for setting up interrupts 1, 2 and 3.

Interrupt functions should generally be designed to return to the main program thread as soon as possible.

For example, to set up interrupt 1 as a counter interrupt using counter 1, and which will be serviced by the C function **my_interrupt()**, you might say something like:

```
// declare some stack space
long my_int_stack[MIN_INT_STACK_SIZE];

// set up the interrupt service function
_set_int_1(CT1, &my_interrupt, &my_int_stack[MIN_INT_STACK_SIZE]);
```

Note that when specifying stack space, you specify the TOP of the stack – i.e. just beyond the last element, not the first element.

Note that interrupts 1, 2 or 3 are all available for use in a non-threaded C program, but that if threading is used in a **NATIVE** C program, interrupt 3 is used to implement the thread support, and should not be used in the user program. However, interrupt 3 CAN be used in a **COMPACT** or **TINY** program, even if that program uses threads. This is because the COMPACT and TINY kernels use time-slicing to implement multi-threading, while the NATIVE kernel uses interrupts.

When in an interrupt service function, all the normal C functionality is available, but threading functions should not be used (because they may block) and also some C functions (such as **stdio** functions) cannot be used because they are not re-entrant. The Catalina HMI functions defined in *hmi.h* can be used instead of **stdio** functions.

Fundamental Interrupt Functions

The fundamental interrupt library functions are defined in the header file *int.h*

They are as follows:

```
enum int_src
```

An enumeration that can be used to specify the source of a specific interrupt. For example, a counter (CT1, CT2, Ct3), a configurable event (SE1, SE2, SE3, SE4) or a pattern (PAT).

```
void _set_CT1(unsigned long CT1);
void _set_CT2(unsigned long CT2);
void _set_CT3(unsigned long CT3);
```

For counter interrupts, set the initial value of the counter.

```
void _add_CT1(unsigned long CT1);
void _add_CT2(unsigned long CT2);
void _add_CT3(unsigned long CT3);
```

For counter interrupts, set the value of the counter that will trigger the interrupt. This can be used within an interrupt service function to specify when the NEXT interrupt should occur.

```
void _set_PAT(int A_or_B, int EQ_OR_NE, unsigned long MASK, unsigned
long MATCH);
```

For pattern interrupts, set the input conditions that must be met to trigger the interrupt.

```
void _set_SE1(unsigned long SE1);
void _set_SE2(unsigned long SE2);
void _set_SE3(unsigned long SE3);
void _set_SE4(unsigned long SE3);
```

For configurable interrupts, set the event that will trigger the interrupt. See the Propeller 2 documentation for how to define each event.

```
void _nix_int_1();
void _nix_int_2();
void _nix_int_3();
```

Cancel an interrupt.

```
void _sim_int_1();
void _sim_int_2();
void _sim_int_3();
```

Simulate (trigger) an interrupt.

```
void _set_int_1(enum int_src SRC, _interrupt SVC, void *stack);
void _set_int_2(enum int_src SRC, _interrupt SVC, void *stack);
void _set_int_3(enum int_src SRC, _interrupt SVC, void *stack);
```

Set up a C function as an interrupt service function for a specific type of interrupt, and using a specified area of RAM as the stack - note that we must point to the TOP of the stack, not the bottom - i,e, just beyond the last element of the stack, not the first element.

```
void _clr_int_1();
void _clr_int_2();
void _clr_int_3();
```

Clear - i.e. disable - an interrupt.

```
unsigned long _get_status();
```

Return the interrupt status bits (actually uses GETBRK wcz). Refer to the Propeller 2 documentation for how to interpret the return value.

```
unsigned long _poll_ANY();
unsigned long _poll_CT1();
unsigned long _poll_CT2();
unsigned long _poll_CT3();
unsigned long _poll_SE1();
unsigned long _poll_SE2();
unsigned long _poll_SE3();
unsigned long _poll_SE4();
unsigned long _poll_PAT();
unsigned long _poll_FBW();
unsigned long _poll_XMT();
unsigned long _poll_XFI();
unsigned long _poll_XRO();
unsigned long _poll_XRL();
unsigned long _poll_ATN();
unsigned long _poll_QMT();
```

Poll for an interrupt of type XXX - return 0 if the event has not occurred, or 1 if it has occurred.

```
unsigned long _wait_ANY(unsigned long timeout);
unsigned long _wait_CT1(unsigned long timeout);
unsigned long _wait_CT2(unsigned long timeout);
unsigned long _wait_CT3(unsigned long timeout);
unsigned long _wait_SE1(unsigned long timeout);
unsigned long _wait_SE2(unsigned long timeout);
unsigned long _wait_SE3(unsigned long timeout);
unsigned long _wait_SE4(unsigned long timeout);
unsigned long _wait_PAT(unsigned long timeout);
unsigned long _wait_FBW(unsigned long timeout);
unsigned long _wait_XMT(unsigned long timeout);
unsigned long _wait_XFI(unsigned long timeout);
unsigned long _wait_XRO(unsigned long timeout);
unsigned long _wait_XRL(unsigned long timeout);
unsigned long _wait_ATN(unsigned long timeout);
```

Wait for an interrupt of type XXX - return 0 if the timeout occurs, or 1 if the event occurs.

```
_cog_ATN(unsigned long cogs);
```

Request the attention of all the cogs whose bits are set in 'cogs'

Interrupt Examples

To build a program manually that uses interrupts, just specify the interrupt library on the command line using the command line option **-lint**. Note that you must also specify **-p2**, since interrupts are only supported on the Propeller 2.

For example:

```
catalina -p2 -lc -lint program.c -C NATIVE
```

If the program uses threads, also specify the thread library. For example:

```
catalina -p2 -lc -lint -lthreads program.c -C COMPACT
```

The *demos\interrupts* folder contains several example programs – one that uses threads and some that does not. Use the **build_all** script to build them, specifying the platform and (optionally) the memory model. For example:

```
build_all P2_EVAL NATIVE
```

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. The **getrealrand()** function will generate 32 bits of true random data.

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 _cogstart_NMM(uint32_t PC, uint32_t SP, void *arg);
```

This function starts a blob that contains an NMM 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_NMM_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_XMM_SMALL(uint32_t PC, uint32_t CS, uint32_t SP, void
*arg);
```

This function starts a blob that contains an XMM SMALL program in a new cog. The C program can accept one 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_XMM_SMALL_cog(uint32_t PC, uint32_t CS, uint32_t SP,
void *arg, unsigned int cog);
```

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

```
int _cogstart_XMM_SMALL_2(uint32_t PC, uint32_t CS, uint32_t SP, void
*arg1, void *arg2);
```

As above, except that the C program can accept *two* void * arguments and will be started in any spare cog.

```
int _cogstart_XMM_SMALL_cog_2(uint32_t PC, uint32_t CS, uint32_t SP,
void *arg1, void *arg2, unsigned int cog);
```

As above, except that the C program can accept *two* void * arguments, and will be started in the specified cog.

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

This function starts a blob that contains an XMM LARGE program in a new cog. The C program can accept one 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_XMM_LARGE_cog(uint32_t PC, uint32_t CS, uint32_t SP,
void *arg, unsigned int cog);
```

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

```
int _cogstart_XMM_LARGE_2(uint32_t PC, uint32_t CS, uint32_t SP, void
*arg1, void *arg2);
```

As above, except that the C program can accept *two* void * arguments and will be started in any spare cog.

```
int _cogstart_XMM_LARGE_cog_2(uint32_t PC, uint32_t CS, uint32_t SP,
void *arg1, void *arg2, unsigned int cog);
```

As above, except that the C 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 _threaded_cogstart_NMM_cog(uint32_t PC, uint32_t SP, void *arg,
unsigned int cog);
```

This function starts a blob that contains an NMM 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 _ctra(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 the counter control bits in the file *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 *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.

propeller2.h, 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 **propeller2.h**, which contains (among other things) the following definitions:

```
extern volatile uint32_t _IJMP3;
extern volatile uint32_t _IRET3;
extern volatile uint32_t _IJMP2;
extern volatile uint32_t _IRET2;
extern volatile uint32_t _IJMP1;
extern volatile uint32_t _IRET1;
extern volatile uint32_t _PA;
extern volatile uint32_t _PB;
extern volatile uint32_t _PTRA;
extern volatile uint32_t _PTRB;
extern volatile uint32_t _DIRA;
extern volatile uint32_t _DIRB;
extern volatile uint32_t _OUTA;
extern volatile uint32_t _OUTB;
extern volatile uint32_t _INA;
extern volatile uint32_t _INB;
```

Once these names are declared as shown above (or **propeller2.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).

In addition to including the file *propeller2.h* (or *prop2.h)*, a program can also include *propeller.h* (or *prop.h)*, which in turn includes *cog.h*, and 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 *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 or 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 to 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 or 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 *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

On the Propeller 2, Catalina provides support for **BlackBox** – a source level debugger with a command-line interface. BlackBox runs under both Linux or Windows, and is included with Catalina.

BlackBox support is enabled using the **–g** (or **–g3**) command-line option, or including that option in Geany.

For more 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 jintly by Bob Anderson and Ross Higson which used the same protocol as BlackBox, but is no longer supported. Use BlackBox instead.

SD Card Support

If you have a Propeller 2 platform that has an SD Card (such as the **P2_EVAL**), 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 use. 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**.

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 inital 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
```

On the Propeller 2 only, support is also provided for a hardware real-time clock using the Parallax RTC add-on board. This is enabled by specifying the Catalina symbol **RTC** (e.g. on the command line). For example, to compile the demo program */demos/examples/ex_time.c* to use the hardware real-time clock, use a command like::

```
catalina -lci ex_time.c -C RTC
```

When using the hardware clock, the time can be set from the C program, but just as with the software clock, it is set only within the plugin and not in the RTC itself, and so just like the software clock, this time will be lost (i.e. revert to the RTC time) when the Propeller is reset. This is largely due to space limitations in the real-time clock plugin, but also for compatibility with the software real-time clock. To tell which clock is in use, a program should check the year on startup - if it is less than 2000 then the software clock is in use. If it is 2000 or later, then the hardware clock is in use (the hardware clock has only 2 digits to specify the year, and the plugin will always report this as year 20xx). Of course, once the year is set by the program, there is no way to tell.

However, separate software is provided external to the clock plugin that allows setting the hardware RTC time, and also various RTC related parameters.

These stand-alone RTC/I2C drivers are provided in the files *rtc_driver.c* and *i2c_driver.c* in the folder *demos\catalyst\time.* There is also a Catalyst **time** utility provided, which uses these drivers to to set the RTC time from the Catalyst command line.

Note that the same program should not use both the RTC plugin and the RTC/I2C drivers, since by default they will both attempt to use the same pins and will interfere with each other.

The method of configuration of the Real-Time Clock pins differs depending on whether a program is using the stand-alone RTC/I2C drivers, or the RTC plugin. Note that a program (such as *demos\catalyst\time\ex_time.c*) may offer the ability to use either option:

- For programs that use the stand-alone RTC/I2C drivers the symbol **RTCBASE** is generally defined in the main program file (e.g. *demos\catalyst\time\time.c*, *demos\catalyst\time\rtc_example.c*, and *demos\catalyst\time\ex_time.c* when using I2C/RTC drivers all define the base

pin to be 24). Edit the files to change the value of **RTCBASE**.

- For programs that use the RTC plugin (such as the demo program *demos\catalyst\time\ex_time.c* when using the RTC plugin option) the base pin is specified by the value of **_RTC_BASE** in the relevant platform include file in the Catalina *target\p2* directory, such as *P2_EVAL.inc*, *P2_EDGE.inc* or *P2_CUSTOM.inc.* Again, the default is base pin 24.

Flash Support

The Propeller 1 required elaborate compiler and loader support to load programs to or/from FLASH RAM, but on the Propeller 2 you can compile any Catalina program to load from FLASH, and there is a simple payload-based script called **flash_payload** that can program the built-in Flash RAM (e.g. on the P2_EVAL board).

You use the **flash_payload** utility in much the same way you use the **payload** loader. For example, to compile and program the othello program to FLASH:

```
catalina -p2 othello.c -lci
flash_payload othello.bin -o2 -b230400
```

The general format of the command is:

```
flash_payload program.bin [ other payload options ]
```

Note that the program name must be the first parameter, the **-o2** option is required, and that you must have the microswitches on the P2_EVAL PCB set correctly to program the FLASH:

FLASH set to ON
P59^ set to OFF
P59v set to OFF

PSRAM and HyperRAM/HyperFlash Support

Catalina supports Roger Loh's 16 Bit PSRAM driver as a Catalina plugin to read and write data to PSRAM on those Propeller 2 platforms that support it, such as the P2 EDGE.

Catalina also supports Roger Loh's HyperFlash/HyperRAM driver as a Catalina plugin to read and write data to the Parallax P2 HyperFlash/HyperRAM add-on on those Propeller 2 platforms that support it, such as the P2 EDGE and P2 EVAL boards.

This section contains only a brief overview of the HyperRAM/HyperFlash support. See Roger Loh's original drivers and documentation (https://forums.parallax.com/discussion/171176/memory-drivers-for-p2-psram-sram-hyperram-was-hyperram-driver-for-p2#latest) for more details.

The configuration parameters for the driver must be specified in the platform files in the *target\p2* directory, such as *P2EDGE.inc* or *P2EVAL.inc*.

These additional memory options can be used *either* as additional storage, *or* as XMM RAM for executing programs too large to fit in Hub RAM (Catalina supports executing XMM programs up to 16 Mb), or as *both* of these.

An example of using PSRAM has been added in *demos\examples\ex_psram.c*, and using Hyper RAM is in *demos\examples\ex_hyper.c*. These programs illustrate *both* uses of PSRAM or HyperRAM.

A real-world example of using PSRAM as external storage is provided in the Catalyst Lua scripting language, which can store and execute Lua programs from PSRAM. See the **Catalyst Reference Manual** for more details.

Note that using PSRAM as external storage is distinct from simply compiling Lua as an XMM program to execute from PSRAM – but this is also possible.

Also note that you cannot currently use both PSRAM and HyperFlash/HyperRAM in the same program.

Using PSRAM and HyperRAM/HyperFlash as additional storage

Using these options as additional RAM storage is enabled by linking with the **psram** or **hyper** libraries (i.e. adding **-lpsram** or **-lhyper** to the Catalina command).

You would compile the PSRAM example program with a command like:

```
catalina -p2 -lc -lpsram -C P2_EDGE ex_psram.c
```

The functions provided in the **psram** library are as follows:

psram_stop

Stop the PSRAM driver.

psram_getResult

Return the result of the last PSRAM driver operation.

psram_read

Do a burst read of multiple bytes from PSRAM.

psram_write

Do a burst write of multiple bytes to PSRAM.

psram_readByte, psram_readWord, psram_readLong

Read a single byte, word or long from PSRAM.

psram_writeByte, psram_writeWord, psram_writeLong

Write a single byte, word or long to PSRAM.

psram_fillBytes, psram_fillWords, psram_fillLongs, psram_fill

Fill multiple PSRAM bytes, words or longs with a specific value.

psram_setFrequency

Configure the PSRAM to suit the specified frequency (only required if the Propeller frequency is changed after startup).

psram_setDelay, psram_getDelay

Set or get the PSRAM delay parameters.

psram_setBurst, psram_getBurst

Set or get the PSRAM burst parameters.

psram_setQos, psram_getQos

Set or get the PSRAM QoS parameters.

psram_getDriverLock

Get the PSRAM driver lock.

Refer to the include file *catalina_psram.h* for more details.

The **hyper** library provides functions similar to those above which can be used to access HyperFlash or Hyper RAM, but with the prefix **hyper_** in place of **psram_**. For example:

hyper_read

Do a burst read of multiple bytes from Hyper Flash/Hyper RAM.

(etc)

The **hyper** library also contains additional HyperRAM & HyperFlash-specific functions, as follows:

hyper_getDriverCogId

get the cog id of the HyperRAM/HyperFlash driver

hyper_getFlags

get the flags associated with a specific HyperRAM/HyperFlash memory

hyper_getDriverLatency

get the driver latency associated with a specific HyperRAM/HyperFlash memory

hyper_getBankParameters

get the bank parameters associated with a specific HyperRAM/HyperFlash memory

hyper_getPinParameters

get the bank parameters associated with a specific HyperRAM/HyperFlash memory

hyper_readRamIR

read Identification Registers associated with a specific HyperRAM memory

hyper_readRamCR

read Control Registers associated with a specific HyperRAM memory

hyper_lockFlashAccess, hyper_unlockFlashAccess

lock or unlock exclusive access to a specific HyperFlash memory

hyper_eraseFlash

erases either a single sector or the entire HyperFlash memory

hyper_programFlash

writes a block of HUB RAM into HyperFlash memory

hyper_programFlashByte

program a single byte into HyperFlash memory

hyper_programFlashWord

program a single word into HyperFlash memory

hyper_programFlashLong

program a single long into HyperFlash memory

hyper_getFlashSize

returns flash size

hyper_getFlashBurstSize

returns flash maximum burst sze

hyper_readReg

read a register from the external memory device

hyper_writeReg

write a register to the external memory device

hyper_readFlashInfo

reads device information of a specific HyperFlash memory

hyper_readFlashStatus

reads the HyperFlash status register of a specific HyperFlash memory

hyper_clearFlashStatus

clears the HyperFlash status register of a specific HyperFlash memory

hyper_readFlashICR

reads Interrupt Configuration Register of a specific HyperFlash memory

hyper_readFlashISR

reads Interrupt Status Register of a specific HyperFlash memory

hyper_readFlashNVCR

reads Non-Volatile Configuration Register of a specific HyperFlash memory

hyper_readFlashVCR

reads Volatile Configuration Register of a specific HyperFlash memory

hyper_writeFlashICR

writes Interrupt Configuration Register of a specific HyperFlash memory

hyper_writeFlashISR

writes Interrupt Status Register of a specific HyperFlash memory

hyper_writeFlashVCR

writes Volatile Configuration Register of a specific HyperFlash memory

Refer to the include file *hyper.h* for more details.

Using PSRAM and HyperRAM as XMM RAM

Using PSRAM or Hyper RAM as XMM RAM is done simply by adding **-C SMALL** or **-C LARGE** to the Catalina command line. For example:

```
catalina -p2 -lc hello_world.c -C P2_EVAL -C LARGE
```

On the **P2_EVAL** Catalina will assume you have a Hyper RAM add-on board, and on the **P2_EDGE** Catalina will assume you want to use the on-board PSRAM. If you want to override this (e.g. to use the Hyper RAM in place of PSRAM on the P2_EDGE) you must also add **-C HYPER**. For example:

```
catalina -p2 -lc hello_world.c -C P2_EDGE -C LARGE -C HYPER
```

Like Roger Loh's PSRAM and Hyper drivers themselves, Catalina's support for them should be considered a beta version until further notice.

Catalina does not currently support using HyperFlash as XMM RAM.

File System Support

If you have a Propeller platform that has an SD or Card (such as the Hybrid), Catalina provides full support for accessing FAT16 or FAT32 file systems on the SD Card[11].

Note that the Propeller 2 can boot from an SD card formatted as FAT32 but not FAT16. However, Catalyst supports both FAT16 and FAT32, so if Catalyst is programmed into FLASH (e.g. using **flash_payload**) and booted from there, programs can be executed from SD cards formatted as either FAT16 or FAT32.

To enable full file system support, simply compile a program with one of the "extended" versions of the standard C library – i.e. the **cx** or **cix** libraries. The default version of the C library (i.e. the **c** library) 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*** *nust 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 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 *fs.h.* These functions are designed to be more space efficient than the standard C functions.

   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 32 kb 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

[11] 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.

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 several options for support for serial devices on the Propeller 2:

1. Use the **TTY** HMI plugin, which configures a single serial port, normally on pins 62 & 63 and at 230400 baud. 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 files specified in the *platform.inc* file.

   On the Propeller 2, the **TTY** HMI option is the default HMI option for all platforms.

2. Use the **SIMPLE** HMI plugin, which configures a single serial port, normally on pins 62 & 63 and at 230400 baud. You use this option by adding **-C SIMPLE** 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 files specified in the *platform.inc* file.

Using the **SIMPLE** serial driver saves a cog over using the **TTY** serial driver, but it may be slower and may not support very high baud rates. However, it supports baud rates up to 230400 and is therefore suitable for use with most serial terminal emulators.

3. Use the **2 Port Serial** library, which allows up to 2 serial ports using one cog, on any pins. You use this option by adding **-lserial2** on the command line or in Geany. This loads the 2 port serial plugin, and links your program with the **libserial2** library. Access to these serial ports are via special library functions, described below. To configure these ports, you can edit the details in **Serial2.spin**.

   Note that by default, the file **Serial2.spin** file configures one serial port (port 0) on pins 62 & 63, and another (port 1) on pins 56 & 57. Both ports are configured for 230400 baud.

4. Use the 8 **Port Serial** library, which allows up to 8 serial ports using one cog, on any pins. You use this option by adding **-lserial8** on the command line or in Geany. This loads the 8 port serial plugin, and links your program with the **libserial8** library. Access to these serial ports are via special library functions, described below. To configure these ports, you can edit the details in **Serial8.spin**.

   Note that by default, the file **Serial8.spin** file configures one serial port (port 0) on pins 62 & 63, and another (port 1) on pins 56 & 57. Both ports are configured for 230400 baud.

Note that you cannot use the 2 port or 8 port serial plugins and the **TTY** or **SIMPLE** HMI plugins in the same program. If you use the 2 or 8 port serial plugin, you should either specify **-C NO_HMI** flag, or specify another HMI option.

The 2 port Serial library (libserial2)

Here are the functions implemented in **libserial2** (also described in the include file *catalina_serial2.h)*. They are designed to be equivalent to the Spin functions defined in the original Spin version of the 2 port serial driver. In all cases, the **port** number is a number in the range 0 .. 1:

int s2_rxflush(unsigned port)

Flush the receive buffer until empty (discard any characters received).

int s2_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 s2_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 s2_rxcount(unsigned port)

Returns the number of characters currently waiting in the receive buffer.

int s2_rx(unsigned port)

Read a byte from the receive buffer. If there are no characters, wait until a character is received.

int s2_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 s2_txflush(unsigned port)

Wait until there are no characters in the transmit buffer (i.e. all bytes have been sent).

int s2_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 s2_txcount(unsigned port)

Returns the number of characters currently waiting in the transmit buffer.

void s2_str(unsigned port, char *stringptr)

Send a null-terminated string to the transmit buffer.

void s2_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 s2_hex(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal.

void s2_ihex(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal, prefixed by a '$' character.

void s2_bin(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary.

void s2_ibin(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary, prefixed by a '%' character.

void s2_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:

s2_dec(port, value)

Send a signed decimal string, up to 10 digits, plus a sign character if required.

s2_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.

s2_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.

s2_putc(port, txbyte)

Same as **s4_tx()**

s2_newline(port)

Send a newline character. By default, the newline is ASCII 10. If the C symbol **S2_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.

s2_strln(port, stringptr)

Send a zero terminated string newline character. By default, the newline is ASCII 10. If the C symbol **S2_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.

s2_cls(port)

Send a Form Feed character (ASCII 12).

s2_getc(port)

Same as **s2_rx()**.

The Multi Port Serial (aka 8 port Serial) library (libserial8)

Below are the functions implemented in **libserial8** (also described in the include file *serial8.h)*. They are designed to be equivalent to the 2 port serial driver functions, but with additional functions to open and close ports manually if required. In all cases, the **port** number is a number in the range 0 .. 7.

Note that there are two ways to open ports when using the 8 port serial driver – either by "autoinitializing" the serial ports (which is the way the current 2 port driver works) or by manually opening and closing the individual ports from C. For a port to be autoinitialized it must have a port number less than **S8_MAX_PORTS** (defined in *serial8.h*) and also have pin numbers in the range 0 .. 63.

The **s8_openport()** and **s8_closeport()** functions have been added so that ports that are not "autoinitialized" (via the *platform.inc* and *serial8.t* in the *target\p2* directory) can be manually configured instead (note that these functions deal with ports, not pins as the spin versions do). Also note that the port number still has to be less than **S8_MAX_PORTS** or the port cannot be opened.

The following is a comment from the original Spin driver about the supported baud rates:

```
The smart pin uarts use a 16-bit value for baud timing which can limit
low baud rates for some system frequencies - beware of these limits
when connecting to older devices.


Baud    20MHz    40MHz    80MHz    100MHz    200MHz    300MHz
------  -----    -----    -----    ------    ------    ------
300       No       No       No        No        No        No
600      Yes       No       No        No        No        No
1200     Yes      Yes       No        No        No        No
2400     Yes      Yes      Yes       Yes        No        No
4800     Yes      Yes      Yes       Yes       Yes       Yes
```

void s8_openport(unsigned port, unsigned baud, unsigned mode,
unsigned rx_pin, char *rx_start, char *rx_end,
unsigned tx_pin, char *tx_start, char *tx_end)

Open the specified port, using the specified baud, mode, pins and buffers. Note that there are two methods of initializing ports – they can be configured in *Catalina_platform.inc and Serial8.spin2*, which is the way the 2 Port and 4 Port serial drivers work. In that case the ports will be configured and opened automatically on startup, using a predefined tx and rx buffer of 64 characters each for each port.

void s8_closeport(unsigned port)

Close the specified port. This can be used on either auto initialized ports or manually initialized ports. However, if an autoinitialized port is closed, the tx and rx buffers it was using will be lost, and new buffers will have to be supplied when it is reopened.

int s8_rxflush(unsigned port)

Flush the receive buffer until empty (discard any characters received).

int s8_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 s8_rxcount(unsigned port)

Returns the number of characters currently waiting in the receive buffer.

int s8_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 s8_rx(unsigned port)

Read a byte from the receive buffer. If there are no characters, wait until a character is received.

int s8_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 s8_txflush(unsigned port)

Wait until there are no characters in the transmit buffer (i.e. all bytes have been sent).

int s8_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 s8_txcount(unsigned port)

Returns the number of characters currently waiting in the transmit buffer.

void s8_str(unsigned port, char *stringptr)

Send a null-terminated string to the transmit buffer.

void s8_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 s8_hex(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal.

void s8_ihex(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in hexadecimal, prefixed by a '$' character.

void s8_bin(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary.

void s8_ibin(unsigned port, unsigned value, int digits)

Send the rightmost **digits** characters of the **value** in binary, prefixed by a '%' character.

void s8_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:

s8_dec(port, value)

Send a signed decimal string, up to 10 digits, plus a sign character if required.

s8_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.

s8_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.

s8_putc(port, txbyte)

Same as **s4_tx()**

s8_newline(port)

Send a newline character. By default, the newline is ASCII 10. If the C symbol **S8_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.

s8_strln(port, stringptr)

Send a zero terminated string newline character. By default, the newline is ASCII 10. If the C symbol **S8_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.

s8_cls(port)

Send a Form Feed character (ASCII 12).

s8_getc(port)

Same as **s8_rx()**.

Cache Support

When using external RAM (such as PSRAM) accessing the RAM can be quite slow. To speed this up, Catalina provides a caching XMM driver to speed up access to the external RAM. 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.

The caching XMM driver can be enabled by defining one of the following symbols on the command line:

CACHED_1K	use a 1k cache
CACHED_2K	use a 2k cache
CACHED_4K	use a 4k cache
CACHED_8K	use a 8k cache
CACHED_16K	use a 16k cache
CACHED_32K	use a 32k cache
CACHED_64K	use a 64k cache
CACHED	use a default size (8K) cache

For example:

```
catalina -p2 hello_world.c -lc -C P2_EDGE -C LARGE -C CACHED_64K
```

Note that the cache requires sufficient free hub memory according to the size of the cache, and also an extra cog. Also note that if the cache is required but not specified, the default cache size of 8K will be used.

Additional cache speed improvements (up to 25% in some cases) can be achieved by defining one or more of the following Catalina symbols:

LUT_PAGE Use the LUT to hold the current cache page and execute XMM code from there instead of from Hub RAM. The page size can be up to 1k (the second 1k of LUT is used as a common code library). The page size limit of 1k constrains the cache geometries that can be used. More details on this are given in *target\p2\constant.inc*

LUT_CACHE Use the LUT to hold the entire XMM cache. This must be combined with CACHED_1K. It gives good performance with a very small cache size, and frees up valuable Hub RAM for other purposes. Only one of **LUT_PAGE** and **LUT_CACHE** can be specified.

CACHE_PINS Use 2 pins in repository mode to communicate between the XMM kernel and the XMM cache, instead of communicating via Hub RAM. The pins used are specified in the platform configuration file (e.g. *P2EDGE.inc*) and cannot be used for any other purpose.

FLOAT_PINS Use 4 pins in repository mode to communicate between the XMM kernel and the Floating point plugin instead of via Hub RAM. The pins used are specified in the platform configuration file (e.g. *P2EDGE.inc*) and cannot be used for any other purpose. **FLOAT_PINS** is only supported by the Float_C plugin, which is the one loaded by default when the XMM kernel is used, or when the -lmc option is specified on the command line.

Unfortunately, no single combination of the above options gives the "best" performance in all possible circumstances - the results depend on the program, the memory model and cache size used, and how much floating point is used by the program.In general, specifying all of **LUT_PAGE**, **CACHE_PINS** and **FLOAT_PINS** gives good results in most cases, and will typically result in a speed increase of around 10% for XMM **LARGE** programs, and 20% for XMM **SMALL** programs.

For example:

```
catalina -p2 startrek.c -lc -lmc -C P2_EDGE -C SMALL -C LUT_PAGE
```

LUT Execution Support

The Propeller 2 has several execution modes:

Cog Execution When the PC is in the range of $00000 and $001FF, the cog is fetching instructions from cog register RAM. This is commonly referred to as "Cog execution mode.".

LUT Execution When the PC is in the range of $00200 and $003FF, the cog is fetching instructions from cog lookup RAM. This is commonly referred to as "LUT execution mode."

Hub Execution When the PC is in the range of $00400 and $FFFFF, the cog is fetching instructions from hub RAM. This is commonly referred to as "Hub execution mode."

All of these execution modes are used by Catalina.

Catalina uses Hub Execution for C code when the **NATIVE** memory model is in use. C code can be in the range $00400 to $7FFFF in **NATIVE** mode.

Catalina uses Cog Execution mode for all plugins, and also for the kernel primitives in all memory models. Kernel primitives use COG addresses $$0000 to $001FF.

Catalina uses LUT Execution mode for common library functions in all memory models. Library functions use LUT addresses $00300 to $003FF.

LUT Execution is useful for implementing functions that require the FIFO, instructions that are not available during Hub Execution, or have very tight timing constraints.

To facilitate the use of the LUT, Catalina leaves addresses $200 to $2FF free, and provides macros that can be used to locate inline PASM code in the LUT, and then call this code from C. See the include file *include/lut_exec.h*, and the demos in the *demos/lut_exec* folder. Here is a brief summary of the LUT Execution macros:

LUT_BEGIN(OFFS, NAME, SIGN)

Define and load (if not already loaded) a LUT function but DO NOT CALL the function.

Parameters:

OFFS - string offset (from $200) to put function (e.g. "$0")

NAME - string name of the LUT function (e.g. "function")

SIGN - string signature of function (e.g. "$DEADBEEF")

Notes:

OFFS must be a PASM constant >= $0 and < $200

NAME must be a unique valid PASM identifier

SIGN must be a PASM constant that is unique for all functions at the specified offset - this is tested by the macro to determine whether the LUT code is already loaded.

LUT_END

Terminate the definition of a LUT function.

Parameters:

None.

LUT_CALL(NAME)

Call an already loaded LUT function.

Parameters:

NAME - string name of LUT function (e.g. "function")

Notes:

No check is made that the LUT function is loaded - whatever code is currently loaded at the offset of the LUT function will be called.

If the LUT function returns a value (in **r0**) it will be returned by this macro - e.g:

```
return LUT_CALL("function");
```

or

```
result = LUT_CALL("function");
```

Using the LUT Execution Macros

Use **LUT_BEGIN** and **LUT_END** to define and load but not call the function, and then use **LUT_CALL** to call the function, either in the same function or in another function with the same parameter profile.

If it is still loaded, the LUT function can be called again using the **LUT_CALL** macro specifying the name of the LUT function.

If they are loaded at different offsets and do not overlap, multiple LUT functions can be loaded at the same time.

A **ret** instruction is automatically appended to the LUT code by the LUT_END macro, so that the LUT function can be invoked by a **call** instruction specifying the LUT function name.

The value left in **r0** by the LUT function will be returned by **LUT_CALL** macro.

For example ...

```
// define load and call the LUT function ...
int function_1(int a, int b) {
   LUT_BEGIN("0", "function_1", "1");
   PASM(
      " mov r0, _PASM(a)\n"
      " add r0, _PASM(b)\n"
   );*
   LUT_END;
   return LUT_CALL("function_1");
}
```

or ...

```
// define and load, but do not call the LUT function ...
load_function_2(int a, int b, int c) {
   LUT_BEGIN("0", "function_2", "2");
   PASM(
      " mov r0, _PASM(a)\n"
      " add r0, _PASM(b)\n"
      " add r0, _PASM(c)\n"
   );
   LUT_END;
}*
```

... and then ...

```
// call an already loaded LUT function ...
int function_2(int a, int b, int c) {
   return LUT_CALL("function_2");
}
```

For C code to be executed from the LUT, the program must be a **NATIVE** program, and the C code must be 'leaf' code - i.e. it must not call other C functions, including C library functions. For example, function_1 above could be written in C as ...

```
// define load and call the LUT function ...
int function_1(int a, int b) {
```

```
    int result;
    LUT_BEGIN("0", "function_1", "1");
    result = a + b;
    LUT_END;
    LUT_CALL("function_1");
    return result;
}
```

If LUT code is to be defined and loaded by one function and then called from another function, both functions MUST have the same parameter profile, local variables and stack frame structure. This is trivial for PASM code, but NON-TRIVIAL for arbitrary C code, so it is recommended that C code always be defined, loaded and executed in the same function.

Note that code to be located in the LUT is limited to 254 longs. The other two longs are used for the signature long (which the macros test to see if the LUT code is already loaded) and the **ret** instruction (which is added automatically by the LUT macros.

Inline PASM executed from the LUT is supported in all memory models, but note that the inline PASM must be normal LUT PASM, not LMM PASM or COMPACT PASM. Also note that only certain cog registers can be used. In general, it is ok to use cog registers **r0** .. **r5**. **t1** .. **t4** and **RI** in any memory model.

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**.
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 -p2 -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

Note that the above command compiles the program in **NATIVE** mode, which is good for speed of execution, but limits the size of Lua programs, since the program and the Lua script it executes must fit into Hub RAM. But other memory models can be used, including **COMPACT** and **XMM** (**SMALL** or **LARGE**).

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

On the Propeller 2, a Catalina *target* is a PASM 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 2 Targets

This section describes the targets provided in the standard Catalina Target Package for the Propeller 2. Six such targets are provided – two each for LMM, CMM and NMM 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 BlackBox 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 NMM target (*nmm_default*). Any program compiled with the **-C NATIVE** command line symbol (or **–x11)** will use this target unless another target is explicitly specified using the **–t** command line option.
- The BlackBox NMM target (*nmm_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 NATIVE** or the **-x11** option).
- The default XMM target (x*mm_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 (x*mm_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 PASM program loaded from the *target\p2* directory, and compiled along with the C program by Catalina. Note that the **_p2** is appended automatically to the **target** directory specified when the **-p2** command-line option is specified.

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 2, the standard Catalina Target Package supports the **P2_EVAL**, **P2_EDGE**, **P2D2** and **P2_CUSTOM** 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):

P2_EVAL	use P2_EVAL pin definitions, drivers and defaults.
P2_EDGE	use P2_EDGE pin definitions, drivers and defaults.
P2D2	use P2D2 pin definitions, drivers and defaults.
P2_CUSTOM	use a user-customized set of pin definitions, drivers and defaults (if applicable)
COMPACT	compile a CMM program (COMPACT memory model)
TINY	compile an LMM program to use the TINY memory model
NATIVE	compile an NMM program to use the NATIVE memory model (Propeller 2 only)
SMALL	compile an XMM program to use the SMALL memory model
LARGE	compile an XMM program to use the LARGE memory model
PSRAM	compile an XMM program to use PSRAM memory
HYPER	compile an XMM program to use HyperFlash/HyperRAM memory
TTY	load a serial HMI plugin with screen and keyboard support.
SIMPLE	load a simple serial HMI plugin with screen and keyboard support.
VGA_640	load a VGA HMI plugin with resolution of 640 x 480.
VGA_800	load a VGA HMI plugin with resolution of 800 x 600.
VGA_1024	load a VGA HMI plugin with resolution of 1024 x 768.
HIRES_VGA	same as VGA_1024
LORES_VGA	same as VGA_640
VGA	same as VGA_640

COLOR_1	use a color depth of 1 bit.
COLOR_4	use a color depth of 4 bits.
COLOR_8	use a color depth of 8 bits.
COLOR_24	use a color depth of 24 bits.
MONO	same as COLOR_1.
NO_KEYBOARD	do not load the keyboard code and one of the USB drivers
NO_MOUSE	do not load the mouse code and one of the USB drivers
P2_REV_A	use instructions supported by the P2 Rev A chip (required for Rev A chips by some HMI drivers).
MHZ_200	use a clock frequency of 200 Mhz (recommended for building Catalyst).
MHZ_220	use a clock frequency of 200 Mhz (required for High resolution VGA).
MHZ_260	use a clock frequency of 260 Mhz (required for High resolution VGA using 4 bit or 8 bit color).
CR_ON_LF	Translate CR to CR LF on output
NO_CR_TO_LF	Disable translation of CR to LF on input
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)
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_16K	Use a 16K cache for XMM access
CACHED_32K	Use a 32K cache for XMM access

CACHED_64K	Use a 64K cache for XMM access
CACHED	Same as **CACHED_8K**

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 P2_EVAL using a with clock support you might use a command like:

```
catalina -p2 test_time.c -lc -C P2_EVAL -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=P2_EVAL CLOCK
```

or as follows (under Linux if using the bash shell):

```
CATALINA_DEFINE="P2_EVAL CLOCK"; export CATALINA_DEFINE
```

Then a command such as:

```
catalina -p2 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   = P2_EVAL 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 2, Catalina uses some of the Propeller 2 Hub resources for specific purposes:

- Some upper Hub RAM is used 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\p2\constant.inc*.
- Counter 1 is used by the **_waitcnt()** function, which is designed to emulate the same function on the Propeller 1 (and also defined as macro **WAIT()** in *propeller.h*).
- Counter 2 is used by the other timer functions (**_waitsec()**, **_waitms()**, **_waitus()**, **_iwaitsec()**, **_iwaitms()** & **_iwaitus()**).
- Interrupt 3 and Counter 3 are used for context switching when a program is compiled to use multi-threading (i.e. **-lthreads** is added to the compilation).

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 512 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 512 kb (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 512 kb. 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).

NMM Support

The Propeller 2 is capable of using the NMM (**N**ative **M**emory **M**odel) mode to support C programs of up to 512 kb. Catalina provides an NMM Kernel for executing such programs.

The memory model used for NMM 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 Native Memory Model is a very simple kernel – all NMM code is natively executed, so the kernel only contains support routines that can be called at run-time to perform certain very common actions – i.e. it is more like a set of subroutines than a true kernel. Native programs are faster and smaller than LMM programs (although still faster than Spin), but larger than CMM programs.

XMM Support

Catalina also provides XMM (e**X**ternal **M**emory **M**odel) support for executing C programs *larger* than 512 kb – but this requires a Propeller equipped with additional RAM hardware.

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. There is an 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 2 platforms or add-on boards, check if there is a file called ***platform*.TXT** or ***board*.TXT** in the *target\p2* directory. For example:

P2EVAL.TXT

P2EDGE.TXT

HYPER.TXT

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 on the Propeller 2. 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.
- **-x2** segments are arranged as **Cnst, Init, Data**, **Code** and the XMM Kernel is used with the **SMALL** address mode.
- **-x5** segments are arranged as **Code, Cnst, Init, Data** and the XMM Kernel is used with the **LARGE** address mode.
- **-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).
- **-x11** segments are arranged as **Code, Cnst, Init, Data** and the NMM kernel is used with the **NATIVE** addressing mode.

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, NATIVE, SMALL** or **LARGE** 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 512 kb of Hub RAM.

SMALL programs 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 512 kb Hub RAM is used only for stack space.

LARGE programs 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, or up to 512 kb on the Propeller 2.

NATIVE On Propeller 2 platforms, Catalina can produce NATIVE programs. These programs can have code and data segments and heap space up to a total of 512 kb in Hub RAM.

NATIVE programs can be loaded using any Propeller 2 program loader (e.g. Payload), or they can be loaded with Catalyst from SD Card.

See the section **A Description of the Catalina Addressing Modes** later in this document for more details on the **TINY, SMALL, LARGE, COMPACT** and **NATIVE** addressing modes.

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 can be used to specify the address to start the **Read/Write** segments (e.g. the data segment). There is usually no need to set this.

-R Read-Only Segment Address. This option can be used to specify the address to start the **Read-Only** segments (e.g. the code segments). There is usually no need to set this.

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
Kernel (CMM, LMM, or NMM)	+ 1
Any HMI plugin	+ 1
SIMPLE driver	+ 0
TTY driver	+ 1
VGA driver	+ 1
USB Mouse drive	+ 1
USB Keyboard driver	+ 1
2 Port Serial driver	+ 1
Float32_A	+ 1
Float32_B (and Float32_A)	+ 2

Float32_C	+ 1
Real-Time Clock	+ 0 / + 1 [12]
SD Card	+ 1
Caching XMM driver	+ 1
PSRAM driver	+ 1
HyperFlash/HyperRAM driver	+ 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:

```
catalina -p2 prog.c -lc
```

When compiling for a platform supported in a different target directory, the equivalent command would be something like:

[12] 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 -p2 prog.c -lc –T"C:\Program Files (x86)\Catalina\My_Target"
```

Target Packages

The standard target package (target\p2)

The **default** Target Package (in subdirectory *target\p2*) can be used with any of the supported Propeller 2 **base** platforms (i.e. **P2_EVAL**, **P2_EDGE, P2D2** etc).

All platform-specific configuration is done via the file *platform.inc*, which includes the appropriate specific platform files. For each supported platform, there is one such platform configuration file (e.g. *P2EVAL.inc*, *P2EDGE.inc*, *P2D2.inc*, *P2CUSTOM.inc*).

This package supports the BlackBox 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 all general-purpose use. Other target packages for specific purposes may be added later.

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 five 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 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

NATIVE PASM (i.e. PASM intended to be executed by the NATIVE Kernel)

pure PASM (i.e. PASM intended to be executed directly by a cog).

While many LMM, CMM or NATIVE PASM 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
  " rdlong r0, ##@abc\n"
  "#else\n"                     // propeller 1 or 2 TINY PASM
  " jmp #LODL\n"
  " long abc\n"
  " rdlong 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.

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.

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. *lmm_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 convenience macro for this function to make it easier to call from C is defined in *propeller2.h*, called **cogstart_PASM**.

A tool to assist in converting a PASM binary into a form suitable for loading from C using **_coginit** directly or via **cogstart_PASM** is provided - this tool is called **bindump**. It is provided in both source and executable form.

A specific example of using **bindump** is provided in the demo program in *demos\p2* called *test_p2.c*. To build it, use the **build_all** batch file provided. E.g:

```
build_all P2_EVAL
```

This batch file does the following commands:

a) compiles the *flash.pasm* PASM program (to produce *flash.bin*):

```
p2_asm flash.pasm
```

b) converts the *flash.bin* to a C include file:

```
bindump flash.bin -c -p 0x > flash.inc
```

c) compiles a C program which loads the resulting binary using **cogstart_PASM**:

```
catalina -p2 -lci  test_p2.c cogutil.c -o test_p2.bin
```

Examine each of the files mentioned above for more detail.

To load and execute the resulting program, simply type:

```
payload test_p2 -i
```

Then follow the instructions.

Write a PASM function that can be called from C

A Catalina C program can call a PASM function directly. This is done by quite a few functions in the Catalina library - look for library sources with a **.e** extension - e.g. *source\lib\catalina\cogid.e*

Precautions when using 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 ‘**:**’ on the Propeller 1, or “.” on the Propeller 2) 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.

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 Catalina symbol reserved for it (e.g. **P2_EVAL**). One symbol (**P2_CUSTOM**) is reserved, and is intended to be modified to suit other platforms.

To modify the details of a **P2_CUSTOM** platform, or add another platform, modify the file *platform.inc* in the *target\p2* directory. This file includes separate files to define the platform constants of each platform. For instance:

P2EVAL.inc

P2EDGE.inc

P2D2.inc

P2CUSTOM.inc

These are all in the target\p2 directory. The default if no platform is specified is to use *P2CUSTOM.inc*, but this can also be changed by editing *platform.inc*.

To use the platform definitions, you define the appropriate Catalina symbol on the command line. For example:

```
catalina -p2 hello_world.c -lc -C P2_EVAL
```

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-lin*k 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).
- **catbind** program in turn then invokes the **p2asm** assembler on the nominated target file (not directly on the binder output). LMM target files are normal Propeller PASM program that refers out to the following objects in other files[13]:
    - The file created by the binder (i.e. the user program);
    - The Catalina Kernel;
    - Any plugins required by the target;

[13] EMM and XMM targets work slightly differently. Refer to the sections on EMM Support and XMM Support.

- **p2asm** assembles the target SPIN/PASM file, and typically produces a binary file (with a **.bin** extension). A listing file is also 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 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:

c 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).

ci **t**he standard C library without floating point support in *stdio* (e.g. in routines such as *printf* and *scanf*). Like **c**, 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 **ci**. If these are required, use one of the other libraries instead. This library is significantly smaller than **c**.

cx 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 **c**, and should only be used where SD card file system access is required.

cix 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 **cx**.

m 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.

ma the standard maths library, with some functions emulated on a separate cog. This library is smaller and faster than **m**, but requires a free cog (this is transparent to the user of the library - the management of the cog is handled automatically by Catalina).

mb the standard maths library, with more functions emulated on two separate cogs. This library is smaller and faster than **ma**, but requires two free cogs (this is transparent to the user of the library – the management of the cogs is handled automatically by Catalina).

mc the standard maths library, with functions implemented using the built-in CORDIC solver of the Propeller 2. This library is smaller and faster than **ma**, and requires only one free cog (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, **c**, **ci**, **cix** and **cx** also provide the standard Catalina HMI functions described in the **HMI Support** section.

Note that there are four complete sets of the above libraries provided with Catalina – one in the *lib\p2\lmm* directory, another in the *lib\p2\cmm* directory, one in the *lib\p2\nmm* directory, and one in the *lib\p2\xmm* directory. They are all functionally identical, but the one used is dependent on the memory model – **TINY** (or **XMM SMALL**), **COMPACT, NATIVE** or **XMM LARGE**.

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, NMM or CMM PASM)
- The Catalina Kernel (PASM)
- Any plugins specified by the Target program (PASM)
- The specified Target program (PASM)

After this comes unallocated space, up to the top of RAM. On the Propeller 2 this is 512 kb or $80000.

On startup, it is the 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. 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. Once the Kernel has been started a plugin must no longer use any Hub RAM space unless that space is below the system break. To make this possible, the Target program can allocate 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-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.

These non-standard memory management functions are defined in the include file *cog.h*, and the various thread functions are defined in *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 Catalina Code Sizes

Given the limited amount of RAM on the Propeller 2 (512 kb) it is easy to generate programs where the binary may initially seem to end up larger than you might expect.

However, Catalina offers many alternatives to reduce the final program size.

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 512 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 -p2 hello_world.c -lc
```

This compiles the program, links it with the **c** version of C89 library and uses the default target and drivers, and the Native kernel. Here are the statistics actually produced by the above command:

```
code = 15184 bytes
cnst = 144 bytes
init = 212 bytes
data = 1080 bytes
file = 25056 bytes
```

At first sight, it appears that even this trivial program consumes over ***20 kb*** of our precious Hub RAM! Don't worry – this can be substantially reduced in various simple ways.

First, it is only the *file* that is 25 kb – and this includes a lot of overheads (we don't need to go into detail here, but these include drivers, kernel code and various types of support code). The actual *program code* is only around 15 kb. And this can be substantially reduced in various simple ways.

Or, you can compile the program as a COMPACT program, which is the most efficient code-wise:

```
catalina -p2 hello_world.c -lc -C COMPACT
```

This compiles the program to use the COMPACT kernel. Here are the statistics actually produced by the above command:

```
code = 9408 bytes
cnst = 144 bytes
init = 216 bytes
data = 1092 bytes
file = 21280 bytes
```

We have already reduced the actual program code to around 9 kb. 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.

We can also choose to use a smaller library. For example, by using the **ci** library (in place of the **c** library) 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 **9kb** to around **3 kb**. To see this, compile the same program using this command instead:

```
catalina -p2 hello_world.c -lci -C COMPACT
```

Here are the statistics:

```
code = 3356 bytes
cnst = 103 bytes
init = 208 bytes
data = 792 bytes
file = 14880 bytes
```

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 -p2 hello_world.c -lci -C COMPACT -ltiny
```

Here are the statistics:

```
code = 1540 bytes
cnst = 15 bytes
init = 224 bytes
data = 792 bytes
file = 12992 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 -p2 hello_world.c -lci -ltiny -C COMPACT -C NO_FLOAT -C
NO_ARGS
```

Here are the statistics:

```
code = 1532 bytes
cnst = 15 bytes
init = 224 bytes
data = 800 bytes
file = 11008 bytes
```

Our code size is now down to around **1.5 kb**, with the total program size (including overheads) is now around **11 kb**. But that's not the end of the story. We can also use the Catalina optimizer:

```
catalina -p2 hello_world.c -lci -ltiny -C COMPACT -C NO_FLOAT -C
NO_ARGS -C NO_EXIT -O5
```

Here are the statistics:

```
code = 1448 bytes
cnst = 16 bytes
init = 212 bytes
data = 0 bytes
file = 10144 bytes
```

Our code size is now down to around **1.4 kb**, with the total program size (including overheads) is now well under **10 kb**.

But even this is not the end of the story. We are still using C library functions that this program doesn't actually *need*. To make the significance of this more evident, consider the following program, very similar to the traditional C “hello, world” program:

```
#include <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 -p2 hello_world_3.c -lci -C NO_FLOAT -C NO_ARGS -C NO_EXIT -C
COMPACT -O5
```

Here are the statistics:

```
code = 84 bytes
cnst = 16 bytes
init = 4 bytes
data = 36 bytes
file = 8576 bytes
```

Our program is now down to 84 bytes of code, and around 8kb of overheads, which (in this case) still includes at least the following:

- the Catalina Kernel (~2 kb);
- a Catalina HMI plugin and drivers (~2.5 kb);

The important thing about this is that once they are loaded into cogs, these do not occupy hub RAM. So this program occupies only ***140 bytes of Hub RAM*** for the code and data.

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 the **c** library, and some do not need to load it at all.

Our final Catalina C program, which implements exactly the same functionality as our original program, may now use as little as 104 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 **p2asm** assembler (on the Propeller 2, Catalina targets are really just PASM 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 p2asm 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 -p2 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 p2asm 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 p2asm 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 $FFFFFE 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.:

$00FFFFFE 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 *demos\loader* directory.

A Note about Self-hosted Catalina

On a suitably equipped Propeller 2, Catalina can be self-hosting. It requires an SD Card and 32MB of PSRAM. This is supported on Propeller 2 platforms with sufficient XMM RAM, such as P2_EDGE with 32MB PSRAM installed (i.e. the P2-EC32MB) or a P2_EVAL with the HyperRAM add-on board.

The *demos\catalyst* folder has a subdirectory called *catalina*, in which a self-hosting version of Catalina can be built. Since it is supported only on a limited number of Propeller 2 platforms, this new demo is not built by default when you build Catalyst. If you have a P2 EDGE board with PSRAM or a P2 EVAL board with HyperRAM then you can build it manually - go to the *catalina* subdirectory, and use the **build_all** script with the same options you used to build Catalyst. For example

```
cd catalina
build_all P2_EDGE SIMPLE VT100 CR_ON_LF USE_COLOR OPTIMIZE MHZ_200
```

or

```
cd catalina
build_all P2_EVAL VGA COLOR_4 OPTIMIZE MHZ_200
```

Note that the **build_all** script will automatically use the **copy_all** script to copy the results to the Catalyst *image* folder.

The self-hosted version of Catalina currently has the following limitations:

- it supports the Propeller 2 only.
- it supports the TINY, COMPACT and NATIVE modes only.
- it does not support the Catalina debugger, optimizer or parallelizer.

- It requires the use of DOS 8.3 file names, because long file names are not supported by Catalyst.

The self-hosted version of Catalina uses the SD card extensively, and this access is quite slow compared to a hard disk, so Catalina now has the option of caching SD card sector reads and writes to improve performance. This cache uses the upper 16MB of the 32MB PSRAM or HYPER RAM, allowing the lower 16MB to be used as normal as XMM RAM.

The cache supports two modes:

Write Through if the sector is in the cache then it is used for reads, but all writes are done both to the cache and to the SD card. This is the default mode.

Write Back if the sector is in the cache then it is used for reads, and all writes are done only to the cache, with the sector in the cache marked as dirty. To write all the dirty sectors in the cache back to the SD card, an explicit cache flush must be performed. It is ok not to flush the cache if the cache has not been filled - the writes will be lost, but the SD card will still be in a consistent state. However, once it has been filled then any sectors not cached will be written directly to the SD card, and so if the cache is NOT subsequently flushed then the SD card can end up in a corrupt or inconsistent state.

The cache is enabled by compiling the library with the **PSRAM** or **HYPER** option specified and then linking the programs that use it with both the extended C library and the appropriate PSRAM or HYPER library (i.e. **-lcx** or **-lcix** and **-lpsram** or **-lhyper**).

Since all programs linked with the library built to use the cache will HAVE to be linked this way, it is recommended that the general version of the catalina library NOT enable the cache, but that a separate version of the library be compiled to use with this option.

For an example, see the **build_psram** and **build_hyper** scripts in the *source/lib* folder, and how these are used by the ***build_all*** script in the *demos/catalyst/catalina* folder.

When compiled as a Catalyst demo, the following catalina-related programs use the cache (in **Write Back** mode) to significantly improve SD card performance:

cpp
rcc
bcc
spp
pstrip
p2asm

For details of the functions supported by the cache, see the file *source\lib\io\sd_cache.h*

Using the cache can reduce the time required for the self-hosted version of Catalina to compile a C program by 20% to 33%.However, even using the cache, Catalina's performance as a self-hosted development system is quite slow, and should be considered a "work in progress".

Here are some typical self-hosted compile times:

othello.c (470 lines) - 10 minutes
startrek.c (2200 lines) - 42 minutes
chimaera.c (5500 lines) - 110 minutes

See the **Catalyst Reference Manual** for details on using the self-hosted version of Catalina.

A Note about DOS 8.3 File names used by Self-hosted Catalina

In order to support the Catalyst requirement to use DOS 8.3 file names, the following include file equivalences are defined. On Windows or Linux, either name can be used, but when using Catalyst on the Propeller 2, the DOS 8.3 file name must be used:

Long File Name	**DOS 8.3 File Name**
catalina_cog.h	cog.h
catalina_commondrv.h	commdrv.h
catalina_float.h	floatext.h
catalina_fs.h	fs.h
catalina_gamepad.h	gamepad.h
catalina_graphics.h	graphics.h
catalina_hmi.h	hmi.h
catalina_hyper.h	hyper.h
catalina_icc.h	caticc.h
catalina_interrupts.h	int.h
catalina_plugin.h	plugin.h
catalina_psram.h	psram.h
catalina_rtc.h	rtc.h
catalina_sd.h	sd.h
catalina_serial2.h	serial2.h
catalina_serial4.h	serial4.h
catalina_serial8.h	serial8.h
catalina_sound.h	sound.h

catalina_spi.h	spi.h
catalina_threads.h	threads.h
catalina_tty.h	tty.h
catalina_vgraphic.h	vgraphic.h
catalina_vgraphics.h	vgraphic.h
mathconst.h	mathcnst.h
propeller.h	prop.h
propeller2.h	prop2.h
smartpins.h	smartpin.h
thread_utilities.h	thutil.h

In addition to the include files, some of the Catalina source code and documentation may still refer to the original long file names used for various files in the target directories - but as these files are only used internally, no equivalences are defined - from Catalina 6 onwards, the DOS 8.3 file names are always used:

Catalina_reserved_null.inc	reserven.inc
Catalina_reserved.inc	reserve.inc
Catalina_defines.inc	define.inc
Catalina_constants.inc	constant.inc
Catalina_arguments.inc	argument.inc
Catalina_platforms.inc	platform.inc
Catalina_plugins.inc	plugin.inc
Catalina_threading.inc	thread.inc
Catalina_blackcat.inc	blackcat.inc
catalina_compact.inc	compact.inc
debug_led.inc	debugled.inc
catalina_default.s	def.t
catalina_blackcat.s	dbg.t
P2_CUSTOM.inc	P2CUSTOM.inc
P2_EVAL.inc	P2EVAL.inc
P2_EDGE.inc	P2EDGE.inc

PSRAM_XMM.def	psram.def
PSRAM_XMM.inc	psram.inc
HYPER_XMM.inc	hyper.inc
Cached_XMM.inc	cached.inc
Catalina_kernel_library.inc	klib.inc
Catalina_thread_library.inc	thlib.inc
Catalina_pre_sbrk.inc	presbrk.inc
Catalina_CMM_kernel_library.inc	cmmklib.inc
Catalina_LMM_kernel_library.inc	lmmklib.inc
Catalina_NMM_kernel_library.inc	nmmklib.inc
unscii-16.inc	font16.inc
unscii-8.inc	font8.inc
unscii-8-fantasy.inc	font8f.inc
unscii-8-thin.inc	font8t.inc
BlackCat_DebugCog.spin2	debugcog.t
Cache.spin2	cache.t
P8X32A_ROM_TABLES.spin2	p1rom.t
Flash_loader_1.2_mod2.spin2	flshload.t
Catalina_Cache.spin2	cogcache.t
Catalina_CogStore.spin2	cogstore.t
cogserial.pasm	cogs2.t
hyperdrv.spin2	coghyper.t
psram16drv.spin2	cogpsram.t
MultiPortSerial.pasm	cogs8.t
Catalina_SD_Plugin.spin2	cogsd.t

Catalina_vga_tile_driver.spin2	cogvga.t
Catalina_1CogKbM_A.spin2	cogkbma.t
Catalina_1CogKbM_B.spin2	cogkbmb.t
Catalina_1CogKbM_Common.spin2	cogkbmc.t
Catalina_1CogKbM_pre_sbrk_A.spin2	kbmprea.t
Catalina_1CogKbM_pre_sbrk_B.spin2	kbmpreb.t
Catalina_Float32_A_Plugin.spin2	floata.t
Catalina_Float32_B_Plugin.spin2	floatb.t
Catalina_Float32_C_Plugin.spin2	floatc.t
Catalina_HMI_Plugin_SIMPLE.spin2	hmisimpl.t
Catalina_HMI_Plugin_TTY.spin2	hmitty.t
Catalina_HMI_Plugin_VGA.spin2	hmivga.t
Serial2.spin2	serial2.t
Serial8.spin2	serial8.t
Catalina_RTC_Plugin.spin2	cogrtc.t
Clock.spin2	clock.t
SDCard.spin2	sd.t
PSRAM.spin2	psram.t
HYPER.spin2	hyper.t
Float.spin2	float.t
HMI.spin2	hmi.t
Catalina_HUB_XMM_Loader.spin2	hubload.t
Catalina_SD_Loader.spin2	sdload.t
Catalina_CMM_library.spin2	cmmlib.t
Catalina_LMM_library.spin2	lmmlib.t
Catalina_NMM_library.spin2	nmmlib.t
Catalina_XMM_library.spin2	xmmlib.t

cmm_progbeg.s	cmmbeg.t
cmm_progend.s	cmmend.t
cmm_default.spin2	cmmdef.t
cmm_blackcat.spin2	cmmdbg.t
emm_progbeg.s	emmbeg.t
emm_progend.s	emmend.t
emm_default.spin2	emmdef.t
emm_blackcat.spin2	emmdbg.t
smm_progbeg.s	smmbeg.t
smm_progend.s	smmend.t
smm_default.spin2	smmdef.t
smm_blackcat.spin2	smmdbg.t
lmm_progbeg.s	lmmbeg.t
lmm_progend.s	lmmend.t
lmm_default.spin2	lmmdef.t
lmm_blackcat.spin2	lmmdbg.t
nmm_progbeg.s	nmmbeg.t
nmm_progend.s	nmmend.t
nmm_default.spin2	nmmdef.t
nmm_blackcat.spin2	nmmdbg.t
xmm_progbeg.s	xmmbeg.t
xmm_progend.s	xmmend.t
xmm_default.spin2	xmmdef.t
xmm_blackcat.spin2	xmmdbg.t
Catalina_CMM.spin2	cmm.t

Catalina_CMM_dynamic.spin2	cmmd.t
Catalina_CMM_threaded.spin2	cmmt.t
Catalina_CMM_threaded_dynamic.spin2	cmmtd.t
Catalina_LMM.spin2	lmm.t
Catalina_LMM_dynamic.spin2	lmmd.t
Catalina_LMM_threaded.spin2	lmmt.t
Catalina_LMM_threaded_dynamic.spin2	lmmtd.t
Catalina_NMM.spin2	nmm.t
Catalina_NMM_dynamic.spin2	nmmd.t
Catalina_NMM_threaded.spin2	nmmt.t
Catalina_NMM_threaded_dynamic.spin2	nmmtd.t
Catalina_XMM.spin2	xmm.t
Catalina_XMM_dynamic.spin2	xmmd.t

Note that the extension **.t** is used instead of **.s** for assembly language source files in the target directory. This prevents name collisions with user programs.

Catalina Development

Reporting Bugs

Please report all Catalina bugs to ross@thevastydeep.com.

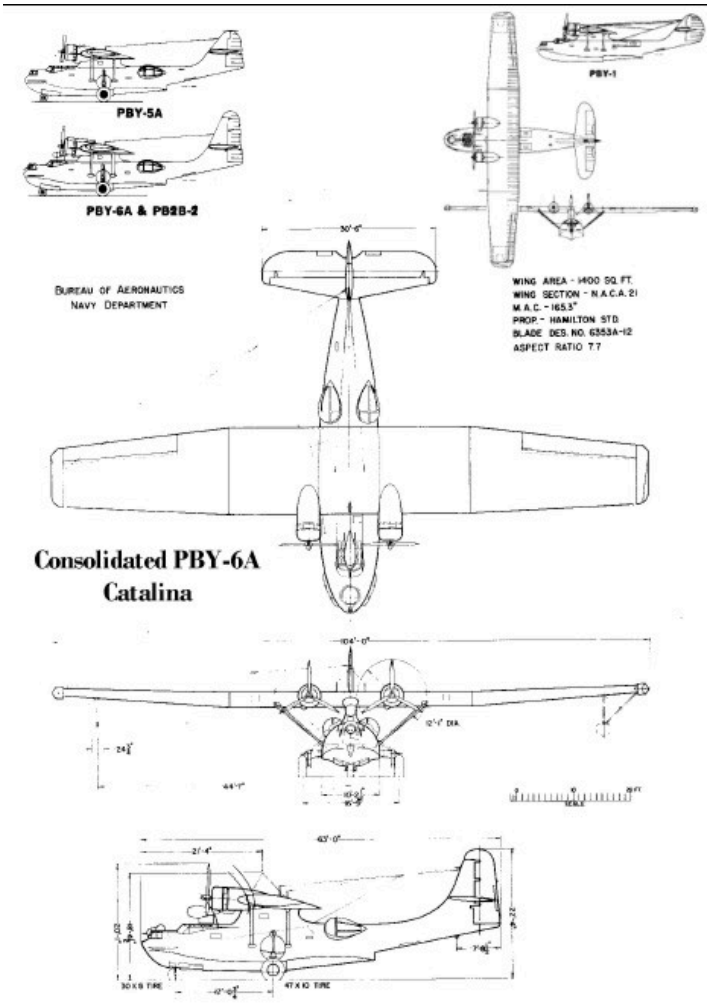
Where possible, please include a *brief* source code example that demonstrates the problem.

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, the Propeller 2 SD card functions, and the 8 port serial driver.

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.
E. R. Smith (Total Spectrum Software) – for the Propeller 2 VGA tile driver.
Garry Jordan (garryj) – for the Propeller 2 USB driver.
Viznut – for the unscii fonts (http://pelulamu.net/unscii).
Roger Loh – for the Propeller 2 PSRAM and HyperFlash/HyperRAM 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 Kernel

The **L**arge **M**emory **M**odel (LMM) kernel was originally developed for the Propeller 1 chip. Although it is no longer necessary on the Propeller 2 (which now has a Native execution mode that can execute programs in a way much more similar to conventional CPUs - see the **Description of the Catalina NMM Virtual Machine** below), for various reasons the LMM Kernel is still available as an option on the Propeller 2.

The Propeller 2 chip is a wonderful thing – the current version has eight 32 bit RISC processors sharing 512 kb of RAM, with all eight processors also sharing 64 general purpose I/O pins! It also has a Native execution mode that allows code to execute from the full extent of RAM. But a significant limitation of the previous Propeller 1 design was that each of the processors (or *cogs*, to use the Parallax terminology) only had direct, full-time access to 2048 bytes of *cog* RAM for the direct execution of Propeller Assembly (PASM) programs. The remaining RAM (referred to as *hub* RAM) was 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 had to be 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 **L**arge **M**emory **M**odel (LMM) mode for the Propeller 1 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 had access to the full 32 kb of hub RAM for the storage of PASM instructions.

LMM mode was essential in enabling traditional high-level compiled languages to be run on the Propeller 1, 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 e**X**ternal **M**emory **M**odel (XMM). For more details on XMM, refer to the **Catalina C Reference Manual for the Propeller 1**.

A Description of the Catalina LMM Virtual Machine

The Catalina LMM Kernel implements 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.

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 (*Branch if Above*) loads the **PC** with the value of its 32 bit *address* 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 (*Branch if Below*) loads the **PC** with the value of its 32 bit *address* parameter if and only if the Propeller's ***C flag is set***. BR_B automatically performs any necessary address translation.

BR_Z * This instruction (*Branch if Zero*) loads the **PC** with the value of its 32 bit *address* parameter if and only if the Propeller's ***Z flag is set***. BR_Z automatically performs any necessary address translation.

BRAE * This instruction (*Branch if Above or Equal*) loads the **PC** with the value of its 32 bit *address* parameter if and only if the Propeller's ***C flag is not set***. BRAE automatically performs any necessary address translation.

BRBE * This instruction (*Branch if Below or Equal*) loads the **PC** with the value of its 32 bit *address* 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 (*Branch if Non-Zero*) loads the **PC** with the value of its 32 bit *address* parameter if and only if the Propeller's ***Z flag is not set***. BRNZ automatically performs any necessary address translation.

CALA * This instruction (*Call Address*) saves the address of the *next instruction following CALA* 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 (*Call Indirect*) saves the address of the *next instruction following CALI* on top of the stack (decrementing **SP** by 4), then loads **RI** into **PC**.

CPYB * This instruction (*Copy Bytes*) 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 (*Division - Signed*) 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 (*Division - Unsigned*) 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 (*Floating Point Addition*) 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 (*Floating Point Comparison*) 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 (*Floating Point Division*) 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 (*Floating Point Multiplication*) 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 (*Floating Point Subtraction*) 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 (*Floating Point from Integer*) converts the integer in **r0** to a floating point value. On return, the result is in **r0**.

INFL This instruction (*Integer from Floating Point*) converts the floating point value in **r0** to an integer. On return, the result is in **r0**.

INIT This instruction (*Initialization*) is the main entry point for the kernel. It is called only once, on startup.

JMPA * This instruction (*Jump Address*) loads the **PC** with the value of its 32 bit parameter. JMPA automatically performs any necessary address translation.

JMPI This instruction (*Jump Indirect*) loads the **PC** with the value of **RI**.

LODA * This instruction (*Load Address*) loads its 32 bit *address* 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 (*Load Frame Reference*) 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 (*Load Long*) 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 (*Multiplication*) 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 (*Pop Many Registers*) 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 (*Push Address*) pushes its 32 bit *address* parameter into the stack (decrementing **SP** by 4). PSHA automatically performs any necessary address translation.

PSHB * This instruction (*Push Bytes*) 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 (*Push Frame Reference*) 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 (*Push Long*) pushes the 32 bit contents of **RI** onto the stack (decrementing **SP** by 4).

PSHM * This instruction (*Push Many Registers*) 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 (*New Frame*) 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 fisrt 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 (*Return from Frame*) 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 (*Read Byte*) 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.

RLNG This instruction (*Read Long*) 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.

RWRD This instruction (*Read Word*) loads the **BC** register with the value of the word in memory pointed to by **RI**.

RETN This instruction (*Return*) 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.

SPEC Perform a special operation that cannot be performed in LMM for various reasons. RI specifies the operation to be performed:

1 => INITCOG

on entry	r0 contains cog id (or ANY_COG) r3 contains pointer to code to load into cog r4 points to data block (REGISTRY, PC, SP)

2 => ROTXY

on entry	r0 contains X r1 contains Y r2 contains angle

3 => CNTHL

on exit	r0 = low 32 bits r1 = high 32 bits

4 => DIV64

on entry	r0 = lower 32 bits of dividend r1 = upper 32 bits of dividend r2 = divisor
on exit	r0 = quotient

SYSP This instruction (*System Plugin*) 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 (*Write Byte*) writes the low order byte of the **BC** register to the byte of memory byte pointed to by **RI**.

WLNG This instruction (*Write Long*) writes the **BC** register to the long of memory pointed to by **RI**.

WWRD This instruction (*Write Word*) writes the low order word of the **BC** register to the word of memory pointed to by **RI**.

Unsupported PASM

The LMM Kernel cannot be used to execute arbitrary PASM instructions. Attempting to execute some PASM instructions would either disrupt the operation of the kernel itself, or fail for other reasons. 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
COGINIT	SPEC with RI=1
QROTATE	SPEC with RI=2
GETCT (to get 64 bit count)	SPEC with RI=3
QDIV	SPEC with RI=4

A Description of the Catalina NMM Virtual Machine

The Catalina NMM Kernel is not really a virtual machine as such, since NATIVE code is simply normal Propeller 2 code executed using Hub Execution (with a few minor exceptions - see **Unsupported PASM** below). But the NMM kernel contains a small set of primitives to provide efficient implementations of some common tasks required by 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 In **NATIVE** mode, the **Program Counter** is the cog Program Counter.

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. In **NATIVE** mode, the **Stack Pointer** is **PTRA**.

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. In **NATIVE** mode, the **Frame Pointer** is **PTRB**.

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.

Primitives

Each Kernel primitive is implemented as a CALLD instruction using PA to call a low (and fixed) address inside the cog that is executing the code. Therefore, like all normal PASM instructions, each NMM primitive occupies a 32 bit long. However, some of the NMM primitives actually occupy *two* 32 bit longs, with a 32 bit long parameter following immediately after the CALLD 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.

CALA * This instruction (*Call Address*) saves the address of the *next instruction following CALA* 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 (*Call Indirect*) saves the address of the *next instruction following CALI* on top of the stack (decrementing **SP** by 4), then loads **RI** into **PC**.

CPYB * This instruction (*Copy Bytes*) 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.

FADD This instruction (*Floating Point Addition*) 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 (*Floating Point Comparison*) 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 (*Floating Point Division*) 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 (*Floating Point Multiplication*) 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 (*Floating Point Subtraction*) 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 (*Floating Point from Integer*) converts the integer in **r0** to a floating point value. On return, the result is in **r0**.

INFL This instruction (*Integer from Floating Point*) converts the floating point value in **r0** to an integer. On return, the result is in **r0**.

INIT This instruction (*Initialization*) is the main entry point for the kernel. It is called only once, on startup.

POPM * This instruction (*Pop Many Registers*) 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.

PSHB * This instruction (*Push Bytes*) 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 (*Push Frame Reference*) 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).

PSHM * This instruction (*Push Many Registers*) 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 (*New Frame*) 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 fisrt 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 (*Return from Frame*) 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).

RETN This instruction (*Return*) 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 (*System Plugin*) 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.

Unsupported PASM

The NMM Kernel executes arbitrary PASM instructions in Hub Execution mode, but attempting to execute some PASM instructions would disrupt the operation of the program. The Catalina Code Generator never generates such instructions – instead, where necessary, it generates the code required to invoke the equivalent NMM 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 NMM equivalent that should be substituted instead:

PASM	LMM Equivalent
RET	RETN (or RETF if NEWF has been called)
JMPRET or CALL	CALA (or load the address in RI and use CALI)

A Description of the Catalina Addressing Modes

On the Propeller 2, Catalina supports five different addressing modes (also known as memory models):

TINY This model is identical to the **TINY** LMM model on the Propeller 1. All addresses are Hub RAM addresses. This memory model can be specified by using the option **-x0** or **-C TINY** on the command line. Note that on the Propeller 2 you would usually want to use the **NATIVE** memory model described below.

COMPACT This model is identical to the **COMPACT** model on the Propeller 1. In this mode, all addresses are 24 bit Hub RAM addresses. This can be specified by using the option **-x8**, or **-C COMPACT** on the command line. Using this option also specifies that the *p2/cmm* library is used, so if you specify all of **-p2**, **-lc** and **-C COMPACT** the library that is actually used will be *lib/p2/cmm/c*

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. Using this

option also specifies that the *p2/lmm* library is used, so if you specify all of **-p2**, **-lc** and **-C SMALL**the library that is actually used will be *lib/p2/lmm/c*

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. Using this option also specifies that the *p2/xmm* library is used, so if you specify all of **-p2**, **-lc** and **-C LARGE** the library that is actually used will be *lib/p2/xmm/c*

NATIVE This model is new for the Propeller 2. In this mode, all addresses are 20 bit Hub RAM addresses. This is specified by using the option **-x11**, or **-C NATIVE** on the command line. This option is the default on the Propeller 2. Using this option also specifies that the *p2/nmm* library is used, so if you specify all of **-p2**, **-lc** and **-C NATIVE**, the library that is actually used will be *lib/p2/nmm/c*

Catalina Calling Conventions

Although only indirectly related to the kernel, 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 "*leaf*" 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 can also make the resulting program much faster to execute.

