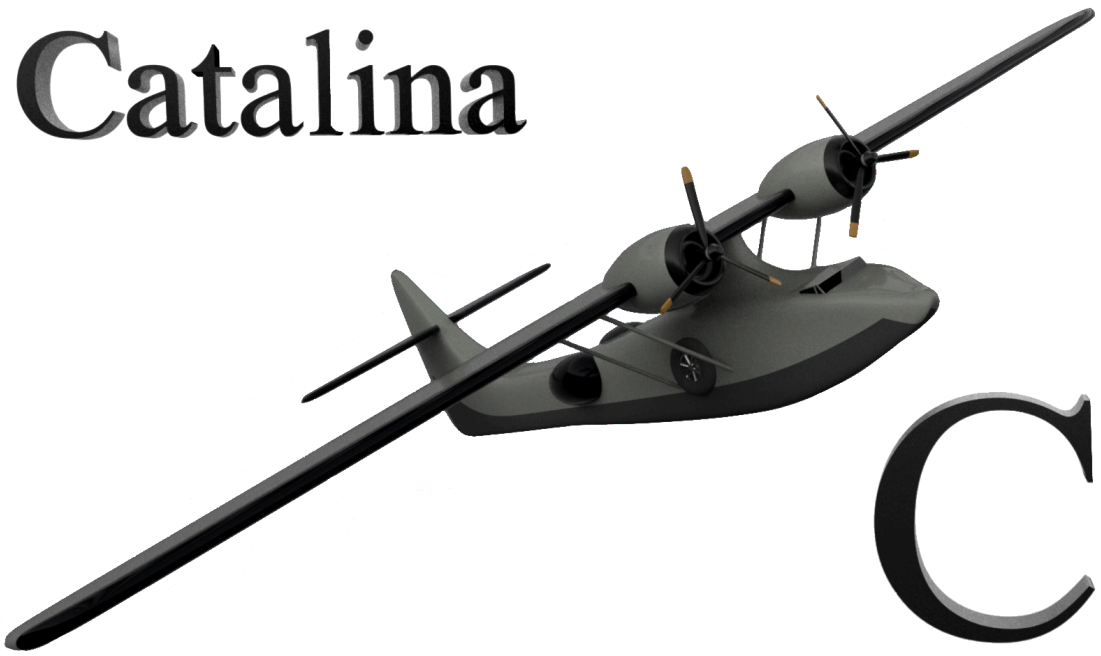

Catalyst
Operating System

Reference Manual

Table of Contents

What is Catalyst?

Catalina is an SD card-based program loader, plus a set of utility programs for the Parallax Propeller. Catalyst supports both the Propeller 1 and the Propeller 2, provided the platform has an SD card. While Catalyst itself does not require XMM RAM, some of the Catalyst applications and utilities do.

When used as intended, Catalyst looks very much like a fully functional Propeller operating system. However, strictly speaking, Catalyst is *just* a program loader – it is not really a true operating system because it does no resource management – however, it can be used to load programs that *do* perform various common resource management tasks – it even comes with a few – e.g. various utilities for doing SD card file management.

While it can be used with any Propeller programs, Catalyst is specifically intended to facilitate loading and using programs compiled with the Catalina C compiler.

Catalyst comes with various applications that can be used for self-hosted Propeller development, including the **vi** text editor, and such tools as a **BASIC** interpreter, a **Pascal** compiler and interpreter, and the **Lua** scripting language. It even comes with a version of Catalina itself, that can be used to compile **C** programs on Propeller 2 platforms with supported PSRAM. It could also be used to edit, compile and then run SPIN programs using the Sphinx SPIN compiler (not included – see http://www.sphinxcompiler.com/).

Features

- Compatible with any Propeller 1 or Propeller 2 platform that supports Catalina and has an SD card available (e.g. Hydra, Hybrid, TriBladeProp, RamBlade, DracBlade, C3, P2_EVAL, P2_EDGE);
- Support for SPIN or Catalina CMM (Compact Memory Mode) or LMM (Large Memory Mode) programs on any supported Propeller 1 platform;
- Support for SMM (SDCARD Memory Mode) and EMM (EEprom Mode) on the Propeller 1, which maximizes the memory available to Catalina CMM and LMM programs on any supported Propeller 1 platform;
- Support for Catalina CMM, LMM or NMM (Native Memory Mode) programs on any supported Propeller 2 platform;
- Support for Catalina XMM (External Memory Mode) programs on any supported Propeller 1 or Propeller 2 platform equipped with external XMM RAM (e.g. Hybrid, TriBladeProp, RamBlade, DracBlade, C3, P2_EVAL, P2_EDGE);
- Support for multi-CPU platforms (e.g. TriBladeProp);
- Support for serial, TV or VGA Human Machine Interface on any supported Propeller 1 or Propeller 2 platform;
- Provides familiar SD card file management commands (e.g. **ls**, **cp**, **mv**, **rm**, **mkdir**, **rmdir**);

- Provides the **vi** full-screen text editor (requires a VT100 compatible terminal emulator if used with a serial HMI, such as Catalina's **payload** program loader executing in interactive mode);
- Supports self-hosted Propeller development (in Pascal, Basic and Lua) on the Propeller 1 and Propeller 2;
- Supports self-hosted Propeller development (in C) on the Propeller 2;
- Supports Lua scripting – Lua scripts can be executed directly from the command line;
- Supports passing command line parameters to both C programs and Lua scripts and to SPIN or SPIN2 programs;
- Supports the YModem serial file transfer protocol, for transferring text and binary files between the Propeller and a PC;
- Supports auto-execution of command scripts on reboot and script execution from the command-line or from C or Lua;
- Supports the 'linenoise' command line editor on the Propeller 2, for command history and command and file name completion.

License

All components of Catalyst are free and open source. Many of the components are licensed under the MIT license. Others are free but are licensed under the GNU General Public License, or other terms and conditions. All licenses are open-source, and free for non-commercial use – however, they are subject to various copyright and other conditions and you should consider the license terms of each component before using any of them in a commercial application.

Installing Catalyst

Overview

Catalyst is now released as part of Catalina, and requires no further installation – but you must still build the appropriate binaries for your Propeller platform.

The Catalyst files will be in the **catalyst** folder. If you installed Catalina in its default location, this will be as follows:

Under Windows: **C:\Program Files (x86)\Catalina\demos\catalyst**

Under Linux: **/opt/catalina/demos/catalyst**

Catalyst Directory Structure

The directory structure for the Catalyst source code should be similar to the following:

```
Catalina
   |
   +--- demos
          |
          + catalyst
          |     |
          .     +--- image
          .     |
          .     +--- catalina
                |
                +--- core
                |
                +--- demo
                |
                +--- dumbo_basic
                |
                +--- fymodem
                |
                +--- jzip
                |     |
                |     +--- doc
                |
                +--- lua-5.4.4
                |     |
                |     +--- doc
                |     +--- etc
                |     +--- src
                |     +--- test
                |
                +--- pascal
                |     |
                |     +--- p5_c
                |     +--- p5_pascal
                |     +--- ptoc
                |
                +--- sst
                |
                +--- xvi-2.51
                      |
                      +--- doc
                      +--- src
```

Building Catalyst from Source

If you have write permission to the Catalina installation folders, you can compile Catalyst in-place. Otherwise, you should copy either the entire demos folder, or just the Catalyst folder, to your own user directory and compile Catalyst there.

The **catalyst** directory contains a **build_all** script that can be used to build Catalyst for all platforms. For example, some commands to build Catalyst would be:

```
build_all HYBRID VT100
build_all TRIBLADEPROP CPU_2 PC VT100
build_all C3 CACHED_1K FLASH TTY VT100 OPTIMIZE
build_all P2_EVAL TTY VT100 MHZ_200
build_all P2_EVAL COMPACT SIMPLE VT100 OPTIMIZE MHZ_200
build_all P2_EVAL LORES_VGA COLOR_4 OPTIMIZE MHZ_200
build_all P2_EDGE SIMPLE VT100 COMPACT OPTIMIZE MHZ_200
```

Building Catalyst and its demo programs requires that you have a version of **make** installed. Linux will usually have **make** installed. If it does not, use the appropriate package manager to install it.

Windows does not have a native version of **make**. The GNU version can be installed either by installing Cygwin, MinGW, MSYS2 or GNuWin32, but the recommended method is to execute the following in a Command Line window (requires an active internet connection):

```
winget install ezwinports.make
```

Note that this installation only has to be done once, but that the current Command Line window will have to be closed and a new one opened for the installation to take effect.

Do not specify a memory model on the command line when compiling Catalyst for a Propeller 1 platform. When compiling for a Propeller 2, you *can* specify a memory model (e.g. **COMPACT**, **TINY**, **NATIVE**) on the command line, but note that even if you do, some parts of Catalyst will always be built using COMPACT mode to save memory (e.g. Lua). If your platform requires a special build, you may need to build each Catalyst component separately.

There is an interactive script called **build_catalyst** which will prompt for arguments. It is intended mainly to be invoked from the Catalina Geany Integrated Development Environment (see the document *Getting Started with the Catalina Geany IDE* for more details) but it can also be used from the command line.

The build_all script copies all Catalyst binaries and example programs to catalyst's *image* folder, ready to be copied to an SD Card.

On the Propeller 1, you can use **payload** to program Catalyst into EEPROM. Note that because the file in the *image* directory has been renamed to be *CATALYST.BIN* (because DOSFS requires MSDOS 8.3 file names) you need to use the **-o1** option to payload, otherwise it will assume the binary is for a Propeller 2 (because it now has a **.bin** extension). For instance, use a **payload** command like:

```
payload -o1 -e catalyst.bin
```

In the same command you can also specify that **payload** open an interactive terminal to interact with Catalyst:

```
payload -o1 -e catalyst.bin -b115200 -i
```

See the **README.TXT** file in the *demos/catalyst* directory for more details, and also the **README** files in each of the subdirectories.

Catalyst HMI options

Catalyst can be built using any of the usual Catalina HMI options supported by the platform, such as **TTY**, **LORES_TV**, **HIRES_TV**, **LORES_VGA**, **HIRES_VGA** etc.

Also, note that just because Catalyst is built to use a specific HMI option does not mean that the programs it *loads* must use the same option, and this includes the

various Catalyst utility programs. For example, Catalyst might be compiled to use a **LORES_VGA** option itself, but then be used to load programs that use **HIRES_VGA**. Or vice-versa.

However, note that some of the Catalyst demo programs work best using either a serial HMI option (e.g. **TTY** or **PC** on the Propeller 1, or **TTY** or **SIMPLE** on the Propeller 2) and a VT100 terminal emulator (such as **payload** or **comms**), or when using the **HIRES_TV** or **HIRES_VGA** options.

Also, memory limitations on the Propeller 1 means there are some specific things to note about the **HIRES** HMI options, as follows.

Catalyst on the Propeller 1 can use the **HIRES_VGA** option. However, Hub RAM limitations mean that Catalyst itself needs to be built as an **EEPROM** program, and some of the Catalyst utilities may only work if they are built as **LARGE** programs.

This means that while Catalyst itself will work in **HIRES_VGA** mode on all platforms, some utilities (such as **cp** & **mv**) may only work in **HIRES_VGA** mode on platforms with XMM RAM.

To facilitate this, two options are available that can be used with the Catalyst **build_all** scripts:

EEPROM_CATALYST	specifies that the Catalyst binary should be built as an EEPROM program.
LARGE_UTILITIES	specifies that the Catalyst utilities should be built as LARGE programs.

Whether you need to specify one or both of these options can depend on the HMI option specified, but also on other options. For instance, if you need to use the cache to access XMM RAM, then you will generally need to use both of these options to build Catalyst in **HIRES_VGA** mode. For example, here is how you might build Catalyst to use **HIRES_VGA** on the C3:

```
build_all C3 FLASH CACHED_1K HIRES_VGA EEPROM_CATALYST LARGE_UTILITIES
```

When you build Catalyst to use **LORES_VGA** or **HIRES_TV** HMI option, you may also find that *catalyst.binary* exceeds the size of Hub RAM and in that case you need to specify the **EEPROM_CATALYST** option, but you may not need the **LARGE_UTILITIES** option – e.g:

```
build_all C3 FLASH CACHED_1K HIRES_TV EEPROM_CATALYST
```

Note that to program Catalyst into EEPROM when the **CATALYST_EEPROM** option is used, you will need to run the **build_utilities** script to build the EEPROM loader, and then use that with payload. For example, to build and load Catalyst to use **HIRES_VGA** on the C3, you might use commands like:

```
cd demos\catalyst
build_all C3 FLASH CACHED_1K HIRES_VGA EEPROM_CATALYST LARGE_UTILITIES
build_utilities
payload -o1 EEPROM ..\bin\catalyst.bin
```

Note that these options are applicable only when using the Catalyst **build_all** scripts and Makefiles – they are not Catalina symbols that can be used in other circumstances (i.e. specifying **-C EEPROM_CATALYST** when compiling *catalyst.c* manually will not have any effect. It is the Makefile that intercepts this symbol and instead uses **-C EEPROM**, but only when building *catalyst.binary*).

On Propeller 2 platforms, and on Propeller 1 platforms where **HIRES_VGA** or **HIRES_TV** is *not* specified, then both Catalyst itself and its utilities will be built as **COMPACT** programs, and do not need either XMM RAM or special loaders.

If you want to use **HIRES_VGA** with Catalyst, but do not have a platform with XMM RAM, you can do so – you may just need to rebuild some of the utilities to use **LORES_VGA**. Just re-run the **build_all** script, and then copy the specific binaries you want to use each time.

Compiling Programs for Catalyst

The main feature that Catalyst adds to existing programs is the ability to run them from the SD card, and to accept command line parameters to be passed to them on startup. This functionality is available both to Catalina C programs as well as SPIN programs.

Compiling C programs to run under Catalyst

Nothing special is required to make C programs run under Catalyst. Any parameters specified on the Catalyst command line will be available to the C program in the normal **argc** and **argv** parameters.

An example program (*demo.c*) is contained in Catalyst's *demo* directory.

To compile the example program as an LMM program, use a command like:

```
catalina demo.c -lci -C C3
```

All Catalyst programs can also be compiled as CMM programs, which substantially reduces their memory size. To do so, use a command like:

```
catalina demo.c -lci -C C3 -C COMPACT
```

For programs that do not need to accept command line parameters, the **NO_ARGS** option can be specified during compilation – e.g.:

```
catalina demo.c -lci -C C3 -C NO_ARGS
```

Of course, in the case of the *demo.c* program, using this option makes the whole program a bit pointless since printing out its arguments is all the demo program does – but for some programs using this option can save a small amount of Hub RAM.

All the above applies also to the Propeller 2, provided you add the **-p2** compiler option. For example:

```
catalina demo.c -lci -C P2_EDGE -C COMPACT -p2
```

Note that Catalina uses. **binary** as the default file extension for binary files on the Propeller 1, but. **bin** on the Propeller 2.

To compile a program to use the SDCARD Memory Mode (SMM) available on the Propeller 1, the executable must have a **.smm** extension, so use commands like:

```
catalina demo.c -lci -C C3 -C SDCARD
mv demo.binary demo.smm
```

The advantage of the SDCARD Memory Mode on the Propeller 1 is that it uses a two-phase loader, with all the plugins and the kernel itself loaded separately from the C program. This means that on LMM platforms, a C program can use all the available Hub RAM (at least up to 31kb) as *application code* space, instead of being limited to what is left after all the plugins and the kernel itself are included (which can be up to 16kb, or *half the available Hub RAM!*). Without SMM mode, the space occupied by these is available as *data* space to the C program, but not as *code* space. No special load options are required on the Propeller 2.

Compiling SPIN programs to run under Catalyst

SPIN and SPIN2 programs can be compiled to be able to access some of Catalyst's special features.

To enable Propeller 1 SPIN programs to interpret command line parameters passed by Catalyst, a special SPIN module is provided called **Catalyst_Arguments.spin**.

A similar module is provided for Propeller 2 SPIN2 programs, called **Catalyst_Arguments.spin2**.

An example program (**demo.spin** for the Propeller 1, or **demo.spin2** for the Propeller 2) is contained in the *Catalyst/demo* directory. To compile them, use a SPIN or SPIN2 compiler such as the Parallax Propeller 2 tool to produce a binary output. On the Propeller 1, you can also use the **spinnaker** spin compiler provided with Catalina:

```
spinnaker demo.spin -b
```

Note that the example program must be manually modified to suit the platform on which you intend to run it – by default the Propeller 1 demo is configured to run using the TV output on a **C3**, and the Propeller 2 demo is configured to use the serial output of the **P2 EVAL** or **EDGE** boards, at 230400 baud. The clock speed, pins and perhaps the TV and video drivers will need to be manually modified to suit other platforms (as is normal for Spin programs).

The **Catalyst_Arguments** module provides three methods:

init(buffer)	*buffer* must point to a buffer of 1200 bytes (300 longs). This method must be called before any of the other argument methods.
argc	this function returns the number of arguments (which may be zero).
argv(i)	this function returns a pointer to a zero terminated string that contains the ***i***th command line argument.

Note that if you run the demo programs but give them no arguments, you will see that argc is 0 (i.e. zero). This is a difference between the C and SPIN/SPIN2 command line processing. Catalyst does not start the CogStore if the program to be executed is given no arguments – but while this is fine for Spin programs (which do not generally expect any arguments anyway) it can cause problems with some C programs, which generally expect argc/argv to have at least *one* argument. So if a Catalina C program finds CogStore is not running on startup, it creates one "null" argument.

Using Catalyst

Using the Catalyst Loader

The main Catalyst binary is intended to be executed in interactive mode whenever the propeller is reset, to allow the interactive entry of commands.

On the Propeller 1 this means the main Catalyst executable is normally loaded into EEPROM. On the Propeller 2 the same effect can be achieved by calling the Catalyst binary *_BOOT_P2.BIX*. Catalyst can also be used in a non-interactive mode as a stand-alone loader for Catalina programs in other Propeller operating systems – see the section titled Catalyst Auto-Execution later in this document for more details on this option.

Note that you can format an SD card as either FAT16 or FAT32 for use with Catalyst, but the Propeller 2 can only boot from an SD card formatted as FAT32, not FAT16. However, if Catalyst is programmed into FLASH RAM (e.g. using **flash_payload**) and booted from there, programs can be executed from SD cards formatted as FAT16 on both the Propeller 1 and 2.

When executed in interactive mode, Catalyst should display a simple banner line similar to the following:

```
Catalyst v8.7
>
```

Where this prompt appears will differ depending on the Propeller platform. For example, on the **RAMBLADE** the Catalyst HMI will by default use a serial terminal emulator running at 115200 baud (e.g. on a PC), on the **C3** it will appear by default on the VGA display, and on the **HYBRID** Catalyst will a local TV display. The **P2_EVAL** and **P2_EDGE** boards will default to using a serial terminal emulator at 230400 baud.

Catalyst may use either a serial terminal, or a local TV or VGA display and keyboard depending on the configuration parameters used when Catalyst is compiled. See the notes on each supported platform given later in this document.

When Catalyst is configured to use the PC serial terminal emulator HMI option, some of the commands expect a VT100 compatible terminal emulator, such as Catalina's **payload** program loader in interactive mode.

At this point, Catalyst commands can be entered. Catalyst contains a few simple *built-in* commands (e.g. **dir**, **cat**, **help**) described below, but most of the Catalyst commands are *external* commands, loaded from the SD card.

Note that whenever the SD Card is removed and reinserted, Catalyst must be restarted by rebooting the Propeller. Otherwise the SD file system access will not work correctly.

Catalyst File Extensions

Catalyst can be used to load and execute normal SPIN binaries as well as Catalina binaries. Generally, such files should be placed on the SD Card with a **.bin** extension (the extension does not need to be specified when executing the program).

However, the main reason for using Catalyst is that it also knows how to load Catalina XMM programs[1], and Catalina SMM programs. On platforms that can support Lua[2], Catalina can also be used to directly execute Lua scripts. Here are the rules Catalyst applies when loading and executing a command:

1. If the command matches the name of a built-in command, that command is executed. Built-in commands are described elsewhere in this document.
2. If the command has a **.lua** or **.lux** extension, or has no extension but matches a file with such an extension, then Lua is loaded and the command is passed to Lua for execution as a Lua script. By default, files with a **.lux** extension will be executed with **LUAX.BIN**, and files with a **.lua** extension will be executed with **LUA.BIN**. If no extension is specified and files with both extensions exist,. **lux** will be used.[3]
3. If the command exactly matches the name of a file, then that file is loaded. Otherwise, if the command has no extension, the loader looks for a filename that matches the command with the following extensions added (in the order shown), and it loads the first one it finds:

   .bin

   .bix (Propeller 2 only)

   .xmm (Propeller 1 only)

   .smm (Propeller 1 only)

   .lmm (Propeller 1 only)

   Catalyst will try two places when searching for each file name:

   The *root* directory of the SD Card;

   The **bin** directory of the SD Card (if one exists);
4. If the file has a **.xmm** extension, it is loaded as an XMM file;
5. If the file has a **.smm** extension, it is loaded as an SMM file;

---

[1] Note that when Catalyst is used to load XMM programs, both Catalyst and the XMM programs must use the same cache and flash options. For instance, if Catalyst is compiled with the options **FLASH CACHED_1K**, then so must all XMM programs (but note that they can be SMALL or LARGE programs). Applies to the Propeller 1 only, and only to XMM programs, and not to LMM, CMM, SMM, Spin or Lua programs.

[2] Executing Lua requires either a Propeller 1 with sufficient XMM RAM, or a Propeller 2.

[3] On the Propeller 2, the program used to execute **.lua** and **.lux** files can be specified by setting environment variables. See the section on Environment Variables for details.

6. If the file has *any other* extension, then on the Propeller 1 it is loaded as a simple binary. On the Propeller 1 it will be assumed to be an LMM (or CMM) file if it is 32 kb or under in size, or an XMM file if it is over 32 kb in size. On the Propeller 2 all programs are loaded the same way (and should just have a **.bin** or **.bix** extension).

Note that a binary file does not need to be a compiled C program – any valid Propeller binary (e.g. a compiled Spin program) can be loaded using these rules.

All commands can have parameters. Lua and C programs know how to process command-line parameters. For Spin programs on the Propeller 1, there is a Spin or Spin2 module provided that can be used to process Catalyst command-line parameters. See the *Demo.spin* or *Demo.spin2* program in the *demos/catalyst/demo* folder.

Catalyst Program Size Limits

Catalyst has a limit for the size of the programs it can load and execute. This is determined by the size of the cluster list and the cluster size itself, and is critical when loading large XMM programs.

The size of the cluster list is set (as **MAX_FLIST_SIZE**) in various places:

- *Catalina_Common.spin* in the Propeller 1 targets (*target/p1*, *embedded/p1* and *minimal/p1*)
- *constant.inc* in the Propeller 2 target (*target/p2*)
- *catalyst.h* in the Catalyst source (*demos/catalyst/core*)

Note that these definitions must all match!

The default setting is for a maximum program size of 4Mb but this will only be achieved when using a cluster size of 32k. For limits when using other cluster sizes, see the table below:

Cluster Size	Max Program Size
512 bytes	64 kbyte
1 kbyte	128 kbyte
2 kbyte	256 kbyte
4 kbyte	512 kbyte
8 kbyte	1 Mbyte
16 kbyte	2 Mbyte
32 kbyte	4 Mbyte

It is generally recommended to format SD Cards as FAT32 and with a cluster size of 32 kilobytes, especially on the Propeller 2. Some of the Catalyst programs, such as Catalina itself, have very large executables that will not be loadable otherwise.

The SDCARD Memory Mode (SMM)

The SMM memory mode is a unique feature of Catalyst. It allows Catalina LMM programs on the Propeller 1 to make much more effective use of the Propeller's limited Hub RAM by loading the program in two phases. This means all plugins/cog programs are loaded first, then the application program is loaded separately – allowing up to 31kb to be used for Catalina application program code. Compare this to normal programs (both Spin and Catalina LMM programs) where up to 16 kb of Hub Ram may be needed simply to hold the programs to be loaded into the various cogs – which makes this space unavailable for use as application program code at run time (it can still be used as data space, but not as application code).

This means that using the SMM memory mode can almost ***double*** the amount of code space available to Catalina LMM programs!

The SDCARD memory model is enabled by defining the **SDCARD** symbol on the command line, and also by saving the program with a **.smm** extension (instead of a .bin extension).

For example:

```
catalina othello.c -lc -C SDCARD
mv othello.binary othello.smm
```

A more detailed description of the SMM load process is given in the file *README.SMM_Loader* in the main Catalina directory.

There are some limitations when using the SMM memory model:

- SMM programs must be 31kb or less (less if they use plugins that allocate buffer space in Hub RAM).
- SMM programs will load the SD plugin even if the program would not normally need it. This may mean the program requires one more cog than usual.

Catalyst Commands

Catalyst Built-in Commands

Some commands are built into the Catalyst loader itself so that they are always available. They will work on any Propeller platform supported by Catalina.

CLS

Clear the screen.

For CLS to work correctly when using a serial HMI, a VT100 compatible terminal emulator (such as **payload** in interactive mode) must be used.

Catalyst External Commands (P1 and P2)

The external commands may be normal SPIN programs or C programs. On Propeller 1 and 2 platforms with sufficient RAM to execute Lua, they may also be Lua scripts. The C external commands will work on any Propeller platform supported by Catalina. They do not require XMM RAM, but they depend on being loaded by Catalyst so that command line arguments can be entered.

HELP

Display some simple help about Catalyst.

LS

list the details of a file or the contents of a directory.

This command accepts wildcards. See the **WildCard Support** section for details.

syntax:

```
ls [options] [file_or_dir] ...
```

options:

```
-h or -?      print help
-l            long format listing
-l -l         very long listing format
-r            recursively list subdirectories
-d            print diagnostics
```

e.g. to list the current (top level) directory:

```
ls
```

or

```
ls /
```

To list directory a/b/c:

```
ls a/b/c
```

For arguments that may be interpreted as files or directories, adding a trailing */* ensures they will be treated as directories. For example:

```
ls bin     -- list just the entry "bin" (if it exists)
ls bin/    -- list the contents of directory "bin" (if it exists)
```

MV

Move one file to another, or one or more files to a directory. If there are only two arguments, and the target does not exist, you must tell mv whether the target is supposed to be a file or a directory.

NOTE: **mv** is essentially a **cp** followed by an **rm** – with the **rm** only performed if the copy succeeds. This means that there must be enough free space to hold two complete copies of the file. It also means that like **cp**, **mv** can be quite slow to execute.

This command accepts wildcards. See the **WildCard Support** section for details.

syntax:

```
mv [options] src [src_ ...] target_file_or_directory
```

options:

```
-h or -?      print help
-f            force overwrite (if target is read-only)
-i            interactive mode
-t            target is a directory
-T            target is a file
-d            print diagnostics
```

e.g:

```
mv a.txt b.txt c.txt my_dir
mv a.txt b.txt
```

In interactive mode, this command will prompt before moving each file. Press **y** or **Y** to process the specified file, **a** or **A** to process ALL files without further prompting, or any other key to skip the specified file and continue with the next file.

RM

Remove one or more files, optionally also removing directories. To remove directories, use **-f**. To remove the contents of a directory as well as the directory itself, use **-r** (recursive).

This command accepts wildcards. See the **WildCard Support** section for details.

syntax:

```
rm [options] file_or_dir ...
```

options:

```
-h or -?      print help
-f            remove directories
-r            recursive remove
-i            interactive mode
-d            print diagnostics
-k            kill (suppress) information messages
-r            recursive removal
```

e.g:

```
rm a/b.txt
rm a.txt b.txt
```

In interactive mode, this command will prompt before removing each file. Press **y** or **Y** to process the specified file, **a** or **A** to process ALL files without further prompting, or any other key to skip the specified file and continue with the next file.

CP

Copy one file to another, or one or more files to a directory. If there are only two arguments, and the target does not exist, you must tell cp whether the target is supposed to be a file or a directory.

This command accepts wildcards. See the **WildCard Support** section for details.

syntax:

```
cp [options] src [src_ ...] target_file_or_directory
```

options:

```
-h or -?      print help
-f            force overwrite (if target is read-only)
-i            interactive mode
-t            target is a directory
-T            target is a file
-d            print diagnostics
```

e.g:

```
cp a.txt b.txt c.txt my_dir
cp a.txt b.txt
```

In interactive mode, this command will prompt before copying each file. Press **y** or **Y** to process the specified file, **a** or **A** to process ALL files without further prompting, or any other key to skip the specified file and continue with the next file.

MKDIR

Make one or more directories, optionally making each parent directory in turn if they do not exist.

syntax:

```
mkdir [options] directory ...
```

options:

```
-h or -?      print help
-p            create parent directories if required
-d            print diagnostics
```

e.g:

```
mkdir a/b/c       <- will make directory c only if a/b exists
mkdir -p a/b/c    <- will make directories a, then a/b,
                     then a/b/c if they do not already exist
```

RMDIR

Remove one or more directories, optionally removing each parent directory recursively (if they are empty).

syntax:

```
rmdir [options] directory ...
```

options:

```
-h or -?      print help
-p            remove parent directories if empty
-d            print diagnostics
```

e.g. to remove directory c (but leave a and b intact):

```
rmdir a/b/c
```

To remove directory a/b/c, then a/b, then a (provided they are empty):

```
rmdir -p a/b/c
```

CAT

Display one or more text files on the display.

The external version of this command accepts wildcards. See the **WildCard Support** section for details.

syntax:

```
cat [options] src [ src ... ]
```

options:

```
-h or -?      print help
-i            interactive mode (paged)
-d            print diagnostics
```

e.g. to display two files, pausing at each page:

```
cat -i file1 file2
```

Enhanced Command Line Editing (P2 only)

On the Propeller 2, Catalyst incorporates the 'linenoise' command line editor to add enhanced functionality. This enhanced functionality is implemented as various keyboard shortcuts, similar to those used by the **ilua** command:

LEFT ARROW (or CTRL B)	move cursor left
RIGHT ARROW (or CTRL F)	move cursor right
UP ARROW (or CTRL P)	previous command in history
DOWN ARROW (or CTRL N)	next command in history
HOME (or CTRL A)	move cursor to start of line
END (or CTRL E)	move cursor to end of line
CTRL U	clear entire line
CTRL K	clear from cursor to end of line
CTRL L	clear screen
CTRL W	clear previous word
CTRL T	swap current and previous characters
CTRL C	exit
CTRL D	at start of line means exit (otherwise it means delete)
TAB	perform command or file name completion

Note that this functionality requires a VT100 compatible terminal emulator. When using payload as the terminal emulator, Line Feed characters should be ignored (by adding **-q2**) and the default attention key is CTRL_A, but it can be set to another key (e.g. by adding **-A24** to set it to CTRL_X). So a suitable fully qualified payload command for use with Catalyst on port 8 at 230400 baud might look something like:

```
payload -i -q2 -A24 -p8 -b115200
```

It is also worth noting that in a Windows Command Line window, CTRL_A may be used to select text. This can be disabled by unselecting the "Enable Ctl key shortcuts" option in the Window properties.

Catalyst External Commands (P2 only)

On the Propeller 2, there are some additional commands, largely to do with the use of environment variables. Environment variables are supported on the Propeller 2 only.

Environment Variables

Environment variables are stored in the file *CATALYST.ENV* in the root directory of the SD Card. This is a text file that can be edited with any text editor (such as **vi**), but unless the format of the file is correct, the results are not defined. Only the first 2048 characters of this file are used. If the definition of an environment variable goes beyond that limit, it will be truncated.

To be able to retrieve the value of an environment variable (e.g from C or Lua using the **getenv()** function), a program must be started using the Catalyst program loader. However, the program does not need to be compiled with the extended C library,

since the environment is passed using CogStore. However, to *set* an environment variable (e..g from C or Lua using the **setenv()** or **unsetenv()** functions) a C program must be compiled with the extended C library (i.e. **-lcx** or **-lcix**).

Note that setting an environment variable does not affect the current environment. The variable is set by writing to the *CATALYST.ENV* file, and will be available to C or Lua programs subsequently loaded by Catalyst. This is not usually an issue when using environment variables in Catalyst itself, since each command involves Catalyst loading a new program, which will always load the new environment variables.

Here are some commonly used environment variables:

PROMPT	The prompt Catalyst should use. Use quotes to include spaces - e.g. **`SET PROMPT="? "`**
TZ	The time zone used by the **TIME** command - e.g. **`set TZ=AEST-10`**
LCCDIR	The base directory used by the **CATALINA** command. The default is blank, which indicates the root directory.
CATALINA_LIBRARY	The path to the libraries used by the **CATALINA** command. The default is **`/lib`**
CATALINA_TARGET	The path to the target files used by the **CATALINA** command. The default is **`/target`**
CATALINA_INCLUDE	The path to the include files used by the **CATALINA** command. The default is **`/include`**
CATALINA_TEMPDIR	The directory used by the **CATALINA** command for temporary files. The default is **`/tmp`**
CATALINA_DEFINE	The default Catalina symbols to define for the **CATALINA** command. For example **`P2_EVAL SIMPLE VT100 USE_COLOR`**
_EXIT_CODE	The exit code of the last Catalyst command executed that set one (**0** indicates success).
_0 .. _22	The last values of command line arguments (**_0** will be the last Catalyst command executed, and **_1** to **_22** will be the values of the command's parameters).
LUA	The program used to execute *.lua* files from the command line. If blank then **LUA** is used. Could also be set to **MLUA**, **XS_LUA** or **XL_LUA**

LUAX The program used to execute *.lux* files from the command line. If blank then **LUAX** is used. Could also be set to **MLUAX**, **XS_LUAX** or **XL_LUAX**

SET

The **SET** command allows environment variables to be displayed or set from the Catalyst command line, or in Catalyst scripts.

The format of the command is:

set [options] [NAME [= [VALUE]] ...]

to display all variables, do not specify a name

to display specific variables, specify names

to unset specific variable, specify **-u** or **name=**

options:

-? or **-h** print a help message

-d print diagnostics

-u unset the specified variable

Values that contain spaces must be quoted. More than one value can be set on the command line. The names of environment variables must start with a letter, number or underscore, and cannot contain the character '='.

For example:

```
set
set A = 1 B = 2
set C="a string value"
set A B C
set -u _EXIT_CODE
set _0=
set PROMPT="? "
set TZ=AEST-11
```

IF

The **IF** command allows testing the value of environment variables and performing actions based on the result. It is primarily intended to be used in Catalyst scripts.

The format of the command is:

if <name> [<action> [<param>]]

or

if <name> <unary_op> [<action> [<param>]]

or

if <name> <binary_op> <value> [<action> [<param>]]

where:

<name> is the name of an environment variable. If only a name is specified, the script exits if the variable does not exist

<unary_op> can be **exists** or **missing** (if no operator is specified, **exists** is assumed).

<binary_op> can be **equals**, **=**, **==**, **!=**, **<>**, **~=**, **<**, **<=**, **=<**, **>**, **>=**. **=>** (note Lua promotes string values to integers if used in an arithmetic expression - so the string "1" is the same as 1)

<value> can be a string or an integer (note that strings that contain spaces must be quoted).

<action> can be **continue**, **exit**, **skip**, **echo**, **add**, **sub**, **set**, **assign** or **prompt** (if no action is specified, **continue** is assumed).

<param> for **skip** this is the number of lines to skip when true (if no parameter is specified then 1 is assumed).

for **echo** this is a value to echo (if no parameter is specified then the value of the variable is echoed).

for **add** and **sub** this is the amount to add or subtract from the variable (a variable that does not exist or which has a value that cannot be interpreted as an integer has value 0).

for **set** this is the value to set in the variable (if no value is specified the variable is deleted).

for **assign** it is the name of another environment variable to assign to the variable.

for **prompt** it is the string prompt to display (if a value is entered it is stored in the variable, otherwise the variable is deleted). If no string is specified the name of the environment variable is used.

For example:

```
if A                    -- exits if variable A doesn't exist
if A continue           --   "
if A exists             --   "
if A missing exit       --   "
if A missing            -- exits if variable A exists
if A exists exit        --    "
if A skip               -- skips one line if variable A exists
```

```
if A missing skip 2          -- skips two line if variable A doesn't exist
if A = 1                     -- exits if value of variable A is not 1
if A = 1 exit                -- exits if value of A is 1
if A = "2" skip 3            -- skips three lines if value of A is 2
if A != "a string" exit      -- exits if value of A is not "a string"
if A echo                    -- echo the value of A if A exists
if A missing echo "No!"      -- echo "No" if A does not exist
if A > 0 sub 1               -- subtract 1 to A if value of A is > 0
if A add 1                   -- add 1 to A if it exists
if A missing prompt ">"      -- prompt for value for A with ">" if A missing
if A exists prompt           -- prompt for value for A with "A=" if A exists
if A missing set "ok"        -- set A to "ok" if A does not exist
if A set                     -- set A to null (i.e. unset it) if A exists
if A missing assign _1       -- set A to the value of _1 if A is missing
```

Note that at least one space must separate each argument, so:

```
if A = 1                     -- will continue if A equals 1
if A=1                       -- will generate an error and exit
if A= 1                      --   "
if A =1                      –    "
```

Note that case is significant in string values but not in the variable names, operations or actions.

Note that although only one condition can be specified in each **IF** command, logical AND or OR of multiple conditions can be achieved using multiple **IF** commands and the **skip** action.

For instance to exit if (A = 1) OR (B = 2) OR (C = 3):

```
if A = 1 exit
if B = 2 exit
if C = 3 exit
```

To exit if (A = 1) AND (B = 2) AND (C = 3):

```
if A != 1 skip 2
if B != 2 skip 1
if C == 3 exit
```

Note that loops can be created using the **IF** command with the **skip** and **add** or **sub** actions. For instance, put the following in a file called *loop* (this file is provided as part of the pre-build demos):

```
@# use argument 1 if provided, otherwise prompt
@if _LOOP missing assign _1
@if _LOOP missing prompt "Number of times to execute? "
@# do loopy stuff here …
@if _LOOP echo
@# ... end loopy stuff
@if _LOOP sub 1
@if _LOOP = 0 skip 2
@# re-execute this script to loop (note use of "_0"!)
@exec _0
@if _LOOP > 0 skip 2
@# clean up after ourselves
@set _LOOP=
```

Then execute the script using the **EXEC** command. For example:

```
exec loop
```

or

```
exec loop 10
```

See the description of the **EXEC** command for more details.

EXEC

The **EXEC** command allows executing a Catalyst script from the command line or from a Catalyst script and passing parameters to it.

The format of the command is:

exec script [parameters ...]

The **EXEC** command works by passing the parameters to the script using environment variables. The Catalyst script name is passed in **_0**, and up to 22 parameters are passed in **_1** to **_22**.

For example, the following command executes the Catalyst script in the file *script* and passes it the value **script** in variable **_0**, the value 3 in variable **_1** and the value "my string" in variable **_2**:

```
exec script 3 "my string"
```

Note that the **EXEC** command does PARAMETER SUBSTITUTION - any arguments passed to it of the form **_0** .. **_22** are substituted with the current value of the relevant environment variable. Note this is done BEFORE the new values are set in these variables. So an example of using exec in a Catalysts script might be:

```
exec _0 _1 "a different string"
```

This would re-execute the currently executing script (whose name is in variable **_0**), passing it the same first parameter as it was passed, but passing "a different string" in place of the second parameter.

The **_n** substitution occurs only in arguments to **EXEC**. But **%n** can be specified anywhere in the script and it will be substituted with the current value of the nth parameter to the current script. In most cases, **%n** can be used wherever **_n** was used, except where **_n** represents the name of an environment variable and not a parameter, such as if it was the name (but not the parameters) of an **if** command.

For an example of **EXEC** and **_n** substitution, see the *loop* example in the description of the new **if** statement. For an example of **%n** substitution, see the *loop2* script in the folder *demos/catalyst/core*, which is a minor variant of the *loop* script.

Note that the exec command *passes control* to the executed script - i.e. it never returns. To call a script from within a script and have it return when that script completes, see the **CALL** command.

See also the **SCRIPT** command, which allows Catalina scripts to be executed from the command line without having to specify **EXEC**.

CALL

The **CALL** command allows a Catalyst script to be called from within another Catalyst script, returning to the original script when the script terminates.

The **CALL** command can also be used on the command line, where it behaves effectively the same as the **EXEC** command - but when used WITHIN a script, **CALL** and **EXEC** behave differently - when **EXEC** is used, the executed script REPLACES the current script, whereas when **CALL** is used, the currently executing script will be resumed once the called script terminates.

For example, to execute the Catalyst script *script_2* from within the script *script_1*, passing it the original parameter **1** and **xxx** as parameters, and then once that completes, call it again with the original parameter **2** and **yyy** as parameters, the script *script_1* might contain lines like:

```
call script_2 _1 xxx
call script_2 _2 yyy
```

If **EXEC** were used instead of **CALL**, the second invocation of *script_2* would never be executed because the first invocation would not return.

Note that calls cannot be nested - i.e. a script can be executed from the command line that calls another script, but that script cannot then call any further scripts.

To support the call capability, two additional lua command have been provided (**_SAVE** and **_RESTORE**) which can save and restore the current values of up to 23 parameters in a single environment variable:

_SAVE	save parameters _0 .. _22 in a named environment variable (**_SAVED_PARAMS** is the default name)
_RESTORE	restore parameters _0 .. _22 from a named environment variable (**_SAVED_PARAMS** is the default name)

The **EXEC** command is still the mechanism to be used for looping scripts, as demonstrated in the *loop* and l*oop2* scripts.

See also the **SCRIPT** command, which allows Catalina scripts to be executed from the command line without having to specify **EXEC** or **CALL**.

SCRIPT

The **SCRIPT** command is provided to allow Catalyst scripts to be executed from the command line without having to specify **EXEC** or **CALL** every time. To use it, copy the file *script.lua* to have the same name as the Catalyst script, but with the extension .**lua**. For example, consider the Catalyst script called *loop* (which is one of the examples provided in the Catalyst pre-built demos). Then to execute the script *loop*, the **EXEC** command can be used as follows:

exec loop 10

But if the **cp** command is used to copy *script.lua* to *loop.lua* - i.e:

cp script.lua loop.lua

Then the **loop** command can be executed without needing **EXEC** - i.e.:

loop 10

ECHO

The **ECHO** command simply echoes its arguments. The format of the command is:

echo [arguments]

For example:

```
echo hello there!
```

or

```
echo %1 %2 %3
```

TIME

The **TIME** command is provided to support the Parallax Real-Time Clock add-on board. It will detect whether or not the RTC add-on board is installed

(by default, on pin 24 - if it is installed on a different pin the time command will have to be recompiled).

It can be used to display both UTC time and local time when the time zone is configured appropriately.

The format of the command is:

time [options] [DD/MM/YY] [HH:MM:SS [AM | PM]]

options:

-? or **-h** print this helpful message and exit

-b mode set battery mode (0 or 1) and report

-c calibrate on RTC start

-r software reset the RTC

-d prompt for new date

-t prompt for new time

-u print UTC time in 12 hour format

-w print UTC time in 24 hour format

-v verbose mode

The time zone is identified using the **TZ** environment variable. If time zone support is *not* required, just set the **TZ** environment variable to “GMT” and set the time in the RTC to the correct local time.

However, it is recommended that the RTC be set to the UTC time and the **TZ** environment variable used to adjust this to the correct local time. The format and use of the **TZ** environment variable is described in:

https://www.gnu.org/software/libc/manual/html_node/TZ-Variable.html

To set the time to one of the time zones hardcoded in the library file *source/lib/time/misc.c* just specify the 3 or more character time zone name.

For example:

```
set TZ=AEST
```

To set the time zone to *any* time zone but WITHOUT corrections for daylight savings, give the time zone name and the offset required to convert that time zone to UTC. For example:

```
set TZ=AEST-10
```

or

```
set TZ=AEST-10:00
```

To set the time zone to *any* time zone WITH corrections for daylight savings, specify a fully qualified TZ. For example:

```
set TZ=AEST-10AEDT,M10.1.0/2,M4.1.0/2
```

The **time** command with no prints the local time in unix format. Command line options can be used to display the UTC time in either the 12 or 24 hour format used by the Parallax RTC driver code.

Lua Scripts

On platforms with enough RAM to execute Lua, Catalyst can execute Lua scripts directly from the command line, which makes it very easy to add new Catalyst commands written in Lua. Some of the Propeller 2 external commands are implemented in Lua (e.g. *if.lua*, *exec.lua, call.lua* and *catalina.lua*) and Catalyst also provides some example scripts designed to demonstrate this functionality:

list.lua – a simple directory listing program (similar to Unix "ls").

find.lua – a simple file searching program (similar to Unix "find").

freq.lua – a simple word frequency counting program.

You invoke Lua scripts from the Catalyst command line just like any other command. You do not need to include the **.lua** extension. For example:

```
list                -- list file details of all files
list *.lua          -- list file details of Lua files
find PRINT *.bas    -- find PRINT statements in all BASIC files
freq *.txt          -- count word frequency in all text files
```

You can also execute other Lua programs directly from the command line. For example:

```
fact                -- execute the factorial example
star                -- execute the Lua version of Super Star Trek
star-tos            -- execute the Lua version of Super Star Trek
                       (using dialog from the original script!)
```

Note that Lua scripts can be compiled to improve load and execution times, and if so they should be given the extension "*.lux*", which tells Catalyst they should be executed using **luax** rather than **lua**.

Wildcard Support

The following Catalyst commands have wildcard support:

ls list files and/or directories

rm delete files or directories

cp copy files

mv move files

ca concatenate and print files

The following wildcard syntax is accepted:

*****	matches zero or more characters
?	matches any single character
[set]	matches any character in the set
[^set]	matches any character NOT in the set where a set is a group of characters or ranges. a range is written as two characters separated with a hyphen, so a-z denotes all characters between a to z inclusive.
[-set]	matches a literal hyphen and any character in the set
[]set]	matches a literal close bracket and any character in the set
char	matches itself except where char is '*' or '?' or '['
\char	matches char, including any pattern character

Some wildcard examples:

Example:	Matches:
a*c	ac abc abbc ...
a?c	acc abc aXc ...
a[a-z]c	aac abc acc ...
a[-a-z]c	a-c aac abc ...

Some actual command examples:

```
ls *.bin *.dat
mv [a-f]*.bin bin
rm ???.dat
cat *.bas
```

Note that wildcards can only be used in the file name portion of a path, not in the directory portion, so you cannot specify an argument like **/b??/*.*** and expect it to match **bin/*.***

There is a Lua program provided (*wild.lua*) which can be used to add wildcard support to any existing program that accepts multiple files as parameters. It requires Lua support and Lua scripting enabled. Just edit *wild.lua* to specify the command. For example, if the command specified in *wild.lua* is "vi" (which it is by default) then

```
luac -o xvi.lux wild.lua
```

will create a command **xvi** which can then be used on the command line to invoke vi on multiple files – e.g:

```
xvi ex*.lua
```

will execute vi on all the Lua example files.

Catalyst Optional Commands

On platforms that contain multiple CPUs (such as the **TRIBLADEPROP**) it may also be convenient to have the normal Catalina multi-CPU utilities loaded onto the Catalyst SD card.

Note that the relevant Catalina utilities are called **CPU_n_Boot.spin** and **CPU_n_Reset.spin**, but if you use the build scripts provided, they may end up being called something like **BOOTn.m** (e.g. **BOOT2.1**).

BOOTn.m

Reboot the Propeller CPU **n**, loading the Catalina Generic SIO Binary Loader so that another program can be loaded (note that this utility is compiled specifically to be run on CPU **m**).

syntax:

```
boot_1
boot_2
boot_3
```

RESETn.m

Just reset the Propeller CPU **n**. Whatever program is loaded into EEPROM will be run (note that this utility is compiled specifically to be run on CPU **m**).

syntax:

```
reset_1
reset_2
reset_3
```

Catalyst Auto-Execution

To facilitate the use of Catalyst as a loader from within other operating systems, Catalyst can check on startup for the existence of the files *EXECONCE.TXT* and *AUTOEXEC.TXT*[4] (in that order) in the root directory of the SD Card. it can process either one only or *more* than one command from these files.[5]

For backwards compatibility, Catalyst can optionally *delete* the AUTOEXEC.TXT file after processing it, but it is now recommended to use the EXECONCE.TXT file if this capability is required.

If the **ONCE** capability is enabled in Catalyst, and the file *EXECONCE.TXT* exists when Catalyst starts, then one command will be read from that file (up to the first zero byte, line terminator or EOF) and the command will be executed as a Catalyst command – including parameters. If the **MORE** capability is not enabled, then the file will then be deleted. If the **MORE** capability *is* enabled, then the command so

---

[4] These are the default names used. They can be modified in *catalyst.c.*

[5] Whether the **AUTO**, **ONCE**, **MORE** and **DELETE** capabilities are enabled can be configured in *catalyst.h. By default, **AUTO** and **ONCE** are enabled, but **DELETE** is not. **MORE** is enabled unless the platform is a Propeller 1 and the **HIRES_VGA** option is specified, in which case there is not quite enough Hub RAM, so to enable **MORE** you may need to disable something else.*

executed is then removed from the *EXECONCE.TXT* file, so that when the Propeller is rebooted, the *next* command will be executed – see the next section for more details on this capability. If the MORE capability is not enabled, the file is simply deleted.

This functionality requires that the SD Card be writable to work correctly.

If Catalyst does not find *EXECONCE.TXT* and the **AUTO** capability is enabled, then Catalyst looks for *AUTOEXEC.TXT*. If it finds that file, it will read one command out of the file (up to the first zero byte, line terminator or EOF) and attempt to execute it as a Catalyst command – including parameters. If the command execution fails, Catalyst will enter interactive mode, otherwise the specified command (which may be any SPIN or C program executable) will be run, with the specified command line parameters. For example, the file may contain a command such as:

```
vi autoexec.txt
```

This would start the vi editor on the *AUTOEXEC.TXT* file itself!

This facility allows other Propeller operating systems to load Catalina LMM, SMM, or XMM programs without having to include the appropriate Catalina loader – Catalyst already incorporates the code to load such programs, and is used strictly as a non-interactive program loader. All the host operating system has to do is create an appropriate *AUTOEXEC.TXT* file and execute **Catalyst** – which is a normal Propeller executable.

When compiling Catalyst, you can optionally enable the **DELETE** capability, which tells Catalyst to delete the *AUTOEXEC.TXT* file after it has been executed once. This capability still exists, although it is now recommended to use the **ONCE** capability instead.

Catalyst Scripts

If the Catalyst **ONCE** and **MORE** capabilities described in the previous section are both enabled, then Catalyst can support very simple scripting.

For example, if you have a file called *COMMANDS.TXT* which contains all the commands you want executed, then executing the Catalyst command:

```
cp command.txt execonce.txt
```

at the Catalyst prompt, or executing the Lua statement:

```
propeller.execute("cp command.txt execonce.txt")
```

will cause all the commands in the file to be executed in sequence. This assumes that each command in the file reboots the Propeller when it has completed.

The **vi** text editor can also be used to create such a command file (which if it is called *EXECONCE.TXT* will be executed as soon as **vi** exits) or the Lua **execute** function can be used to easily add multiple commands to the file, one per line. For example, executing the Lua statement:

```
propeller.execute("vi abc\n vi def")
```

will cause the propeller to first reboot and execute the command **vi abc**, and then when vi exits, the propeller will reboot and execute the command **vi def**.

Note that this capability is enabled by default on both the Propeller 1 and the Propeller 2, but on the Propeller 1 there may not be enough Hub RAM available, depending on the HMI in use and other options selected. You may need to disable either the **MORE** capability, or the **LUA** capability that allows Lua commands to be executed directly from the command line (you can still execute them by specifying them to Lua – e.g. by entering a command like **lua list.lua** instead of just **list**).

Note that when a script is executing, rebooting the Propeller will abort the command being executed and restart Catalyst, but when Catalyst restarts it will resume the script and execute the next command. To terminate an executing script, hold down any key while rebooting the Propeller. A message like the following will appear:

```
Continue auto execution (y/n)?
```

Press **Y** to continue the script, or **N** to terminate it.

To assist in writing scripts, lines whose first character is a **@** are not echoed (but the **@** is otherwise ignored) and lines with **#** as the first character are treated as comments and not executed. The two can be combined as **@#**

For instance:

```
# this line will be echoed but not executed
@# this line will be neither echoed nor executed
# the following command will be not be echoed but will be executed:
@ls
# the following command will be both echoed and executed:
ls
```

If enabled, the scripting capability applies to both the AUTOEXEC.TXT and EXECONCE.TXT files. However, unlike the EXECONCE.TXT file, when the last command in the AUTOEXEC.TXT file has been executed, the entire file will then be re-executed.

On the Propeller 2, the Catalyst scripting capability is significantly enhanced by the **SET**, **IF**, **EXEC**, **CALL**, **ECHO** and **SCRIPT** external commands. See the description of those commands for details.

Catalyst Applications

Catalyst provides a rich set of application programs. However, all the example applications provided require XMM RAM to run on a Propeller 1. Some will run only on platforms with 512k of XMM RAM or more installed. On the Propeller 2, which has 512kb of Hub RAM, all the application programs should run normally.

This section does not describe each application in detail – it only describes how to run the application programs from the Catalyst command line. See the individual application program documentation for more details on the application itself.

DUMBO BASIC

Load the Dumbo BASIC interpreter. Note that Dumbo BASIC is just an interpreter – the programs must be created externally (e.g. using the vi text editor).

syntax:

```
dbasic [ basic_program.bas ]
```

e.g:

```
dbasic eliza.bas
```

If no parameter is specified, dbasic will prompt for the name of the basic file to execute.

LUA

Load the Lua interpreter (**lua**), the Lua execution engine (**luax**) or run the Lua compiler (**luac**).

syntax:

```
lua [script [ parameters ] ]
luac -o output_filename script
luax [script [ parameters ] ]
```

e.g:

```
lua fact.lua
luac -o f.out fact.lua
luax f.out
```

If no file is specified to the **lua** command, commands can be entered directly on the terminal.

The Lua interpreter (**lua**) can be used to execute both text lua programs, or compiled lua programs. It can also be used interactively if no parameters are specified.

The Lua execution engine (**luax**) does not load the Lua parser, and so it can only be used to execute lua programs where all the Lua code to be executed is pre-compiled.

The Lua compiler (**luac**) compiles a Lua program to byte code, which speeds up loading – but the resulting file can be executed with **lua** or with **luax**.

*NOTE: when using **luac**, if you omit the **–o** parameter (which must be the first parameter) then **lua** will output the binary result to the file **luac.out**.*

NOTE: Just because an example Lua program is included does not guarantee it can be executed – on the Propeller 1 this depends on how much XMM RAM is available, and some of the Lua examples will not execute correctly.

ILUA

Load the Lua interpreter with a more functional Read-Eval-Print Loop, based on 'lua-repl' and 'linenoise'.

syntax:

```
ilua
```

The **ilua** command accepts no parameters. It is itself a lua script (*ilua.lua*) which is executed by the normal lua command interpreter, but which loads various plugins to add enhanced functionality. This additional functionality is implemented as various keyboard shortcuts:

LEFT ARROW (or CTRL B)	: move cursor left
RIGHT ARROW (or CTRL F)	: move cursor right
UP ARROW (or CTRL P)	: previous command in history
DOWN ARROW (or CTRL N)	: next command in history
HOME (or CTRL A)	: move cursor to start of line
END (or CTRL E)	: move cursor to end of line
CTRL U	: clear entire line
CTRL K	: clear from cursor to end of line
CTRL L	: clear screen
CTRL W	: clear previous word
CTRL T	: swap current and previous characters
CTRL C	: exit
CTRL D	: at start of line means exit (otherwise it means delete)
TAB	: command completion

NOTE: Just because ilua is included does not guarantee it can be executed – on the Propeller 1 this depends on how much XMM RAM is available.

MLUA

mLua is a version of Lua with Catalina's **threads** multi-processing module compiled in. Size limitations mean **mLua** is supported only on the Propeller 2. For more details, refer to the document **Lua on the Propeller 2 with Catalina**.

Load the Multi-processing Lua interpreter (**mlua**), or the Multi-processing Lua execution engine (**mluax**).

syntax:

```
mlua [script [ parameters ] ]
```

```
mluax [script [ parameters ] ]
```

e.g:

```
mlua ex1.lua
mluax e.lua
```

If no file is specified to the **mLua** command, commands can be entered directly on the terminal.

The **mLua** interpreter (**mlua**) can be used to execute both text lua programs, or compiled lua programs. It can also be used interactively if no parameters are specified.

The **mLua** execution engine (**mluax**) does not load the Lua parser, and so it can only be used to execute lua programs where all the Lua code to be executed is pre-compiled.

Note that there is no specific **mLua** compiler – the normal Lua compiler (**luac**) can also be used to compile **mLua** programs.

JZIP

Load the JZIP Infocom game interpreter. The number of rows and columns to use for the screen size can be specified on the command line, but the game will detect the actual screen size when local devices are used, and assume 80×24 when using the PC HMI option.

syntax:

```
jzip [-ccols] [-lrows] [ game.dat ]
```

e.g:

```
jzip zork1.dat
```

If no parameter is specified, jzip will prompt for the name of the game file to execute.

PASCAL

Load the Pascal interpreter (**pint**), or run the Pascal compiler (**pcom**).

syntax:

```
pint [compiled_program]
pcom [ program_to_compile [compiled_program] ]
```

e.g:

```
pint startrek.p5
pcom startrek.pas startrek.p5
```

The output files from the compiler must be executed with the interpreter.

If no file is specified to the **pcom** or **pint** commands, the programs will prompt for a file name.

By convention, the compiled version of the Pascal program **prog.pas** is normally called **prog.p5**

In addition to a few sample programs, two precompiled programs are provided – **startrek.p5** and **basics.p5**. The first is yet another version of the classic Star Trek game and the second is a basic interpreter. These programs are also provided in compiled form because they can each take several hours to compile on the Propeller 1 – even loading the precompiled programs can take a minute or two.

SUPER STAR TREK

Play a game of Super Star Trek.

syntax:

```
sst
```

There are no parameters to this command. See the document sst.doc for help.

VI

Load the XVI text editor (the binary is renamed to **vi** for convenience). The program accepts various options – see the xvi documentation for more details.

syntax:

```
vi [options] [ filename ... ]
```

e.g:

```
vi sample1.txt
```

A common option to specify is **–s format=msdos** or **–s format=unix** to specify whether msdos or unix line termination is to be used (the default is unis, so if you open an msdos file you may see extraneous **^M** characters at the end of each line).

If more than one filename is specified, vi will open the first two in separate windows. After that, use **:n** (i.e. *colon n*) to move to the next file.

To exit vi, use **:q** (i.e. *colon q*). If you have modified the file but want to quit anyway, use **:q!** (i.e. *colon q exclamation mark*).

Within vi you can type **:help** to get help, and **:clos**e to close the help window.

YMODEM

Use the Free YModem program, which implements the YMODEM serial file transfer protocol to send text or binary files between the Propeller and the Host PC. When Catalyst is built, it will build four executables – a stand-alone **send** and **receive** program for the Propeller, and a stand-alone **send** and **receive** program for the Host PC (either Windows or Linux).

Syntax (on the Propeller or the Host PC):

```
send     [ options ] file

receive  [ options ] [ file ]
```

options:

```
-h or -? print a helpful message and exit (-v for more help)
-b baud  use specified baudrate
-B baud  same as -b
-d       diagnostic mode (-d again for more diagnostics)
-p port  use port
-s msec  small/slow mode - use 128 byte packets, msec char time
-t msec  set general timeout in milliseconds
-v       verbose mode (and include port numbers in help)
-x       no exit mode (e.g. to read output)
```

e.g. on the Host:

```
send my_file.bin -s0 -b115200 -p11
receive -v -b 115200 -p11
```

e.g. on the Propeller:

```
send my_file.txt
receive
```

The main difference in options between the **send** and **receive** programs are that file name is required for send, but is optional for **receive,** and the **-s** option applies only to the send program.

The main difference in options between the Propeller and Host versions is that the Propeller versions cannot specify the baud rate 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*).

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.

The Propeller **send** and **receive** programs will be copied to the *demos/catalyst/bin* directory by the Catalyst **build_all** script, but the Host PC executables are left in the *demos/catalyst/fymodem* directory. You may want to manually copy them to Catalina's *bin* directory, or to the directory in which they will be used.

Note that by default, the build_all script builds the Propeller executables with the **-C NO_HMI** option, because if a serial HMI option is in use it would interfere with the YMODEM protocol which generally uses the same serial port as the serial HMI. However, if your Propeller uses another HMI option you

can edit the Makefile to remove the **-C NO_HM**I option with your own HMI option (e.g. **-C VGA**). Without this, the **-v -d** and **-h** command line options, which control the messages generated by the program, are a bit useless.

Typically, you use these by starting the Propeller end first (either **send** or **receive**) and then the corresponding PC program (i.e. **receive** or **send**).

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 sending 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 **-s**5, -**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**, or another terminal emulator that supports YMODEM (such as **minicom** on Linux).

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.

When using a terminal emulator that has YMODEM protocol support[6], you typically execute the **send** or **receive** on the Propeller first, and then initiate the YModem transfer in the terminal emulator. For examples of using payload, see the section on **Catalina YModem Support** section in the **Catalina Reference Manual**.

CATALINA

Catalyst supports a self-hosted version of the Catalina C compiler. This is only 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.

If it is installed, the self-hosted version of Catalina will be contained in the following directories:

bin	Catalina executables
include	C include files
lib	C libraries
target	Catalina runtime support files
tmp	a directory used to store temporary files

The executable files in the *bin* directory will include:

catalina.lua	a Lua version of the PC **catalina** command, which manages the compilation process.
cat_env.lua	a Lua program that does the same job as the PC **catalina_env** command to display the Catalina environment.
binbuild.bin	a utility to combine the compiled C program files and target files when they are compiled in SMM, EMM or XMM mode, or using Quick Build. Can be invoked stand-alone, but is normally used internally by Catalina.
binstats.bin	a utility to print statistical information about a binary file (code size, data size etc). This can be invoked stand-alone, and is also used internally by Catalina.
cpp.bin	C preprocessor. This is used internally by Catalina.
rcc.bin	C compiler. This is used internally by Catalina.
bcc.bin	Catalina binder and library manager (see the separate description of this command, below)

[6] For the Propeller 1, the terminal emulator must use 128 byte blocks, and not assume it can use 1024 byte blocks – i.e. it must be able to use YMODEM and not just YMODEM-1K. Some programs (e.g Tera Term, ExtraPuTTY) assume they can always use 1024 byte blocks. If you use the payload terminal emulator you can choose to use either.

spp.bin	PASM preprocessor. This is used internally by Catalina.
p2asm.bin	PASM assembler (see the separate description of this command, below).
pstrip.bin	a utility to reduce the size of PASM files. This is used internally by Catalina.
xl_vi	The vi text editor for very large files (the normal Catalyst vi is fine for editing most C source files, but large C source files (such as *chimaera.c*) or the intermediate files generated by Catalina can be too large for it, so this XMM LARGE version is also included.

The Lua version of the **catalina** command supports many but not all of the command-line options of the PC version. Use the command **catalina -?** to see what options are currently implemented.

You can compile C programs at the Catalyst prompt using the **catalina** command, just as you would on a PC.

A selection of C demo programs is included. These programs are the same as the PC-based Catalina demo C programs, but in some cases the names have been changed to meet the Catalyst file naming limitations (i.e. to use DOS 8.3 file names):

hello.c	the classic C "Hello, World" program.
pintest.c	a demo program to toggle an LED.
my_prog.c & my_func.c	programs used to demonstrate libraries (see the description of the **bcc** command)
othello.c	the othello (aka reversi) game.
startrek.c	the classic Star Trek game.
chimaera.c	an adventure game. Note that you will need to use **xl_vi** to edit this file - it is too large for the normal Catalyst **vi** program.
diners.c	a demo of Catalina multi-threading.
station.c	a demo of POSIX pthreads.
intrrpt.c	a demo of interrupt handling.
psram.c	a demo of using PSRAM.

You can compile the demo programs using the following commands:

```
catalina hello.c -lci -v
catalina othello.c -lci -v
catalina startrek.c -lc -lmc -v
catalina chimaera.c -lcx -lmc -v
```

```
catalina diners.c -lci -lthreads -v
catalina station.c -lci -lthreads -v -C NO_REBOOT
```

Note that some of the demo programs require the platform and/or specific libraries to be specified to compile correctly:

```
catalina pintest.c -lci -C P2_EDGE -v
catalina psram.c -lci -lpsram -C P2_EDGE -v
catalina intrrpt.c -lci -lthreads -lint -C P2_EDGE -v
```

or

```
catalina pintest.c -lci -C P2_EVAL -v
catalina psram.c -lci -lhyper -C P2_EVAL -v
catalina intrrpt.c -lci -lthreads -lint -C P2_EVAL -v
```

The **-v** flag is optional in all cases, but is recommended.

The **-C CR_ON_LF** option can be added when using a VT100 terminal emulator. It is recommended that you instead adjust the settings of your terminal emulator to implicitly execute a CR every time it receives an LF. Payload, Comms, PuTTY, and Tera Term all have such a configuration option. But if your terminal emulator does not, you can use this option instead.

NOTE: Catalyst has a limit of 23 arguments that can be specified in a single command, including in a command script. If this number is exceeded when generating the compilation script, the catalina command will print the message "**too many command line arguments - try using CATALINA_DEFINE"** and stop. This will usually be because there are too many options being specified. Instead of specifying so many arguments on the command line, put the definition of Catalina symbols (i.e. **-C** options) into the **CATALINA_DEFINE** environment variable.

NOTE: Catalina has very large executable programs (**rcc** is over 2.5 Mb), which means that it will only be executable on an SD card formatted as FAT32 and with a cluster size of 32Kb (see the section of this document titled **Catalyst Program Size Limits**).

WARNING: The larger C demo programs can take a long time to compile. Even the standard C "hello world" program (*hello.c)* takes almost 10 minutes. Compiling the chimaera game (*chimaera.c*, approx 5,500 lines of C code) takes around 2 hours. This is why the **-v** option is recommended - without it, there is no way of telling that anything is actually happening until the compilation completes.

The self-hosted version of Catalina supports the Quick Build option (enabled by specifying **-q** on the command line or defining the Catalina symbol **QUICKBUILD**) that can reduce compilation times, but comes with some limitations. Refer to the **Catalina Reference Manual** for the Propeller 2 for details on using Quick Build.

All the C demo programs are copied to the root directory, but it is worth noting that neither the source nor the output needs to be in this directory. They could, for example, be in a *demos* subdirectory. However, note that Catalina has no concept of a 'current' directory, and will generally store its output in the same directory as the source it is compiling. So, for instance to compile the file hello.c if it was in a directory called *demos*, a command like the following might be used[7]:

```
catalina -v demos/hello.c -lci
```

This command would leave its output (hello.bin) in the demos directory. This means that to execute the output, the whole file name including the extension has to be entered, since Catalina only automatically tries adding various extensions to things that look like commands, and those must be either in the root or bin directories.

So to execute it from the demos directory, use the following command:

```
demos/hello.bin
```

To store the output in the root or bin directories instead of the demos directory, add an explicit -o option to the catalina command. For example:

```
catalina -v demos/hello.c -lci -o hello
```

or

```
catalina -v demos/hello.c -lci -o bin/hello
```

Now when the compilation is complete, the result will be in the root or bin directory, and can be executed by just entering the command:

```
hello
```

Note that the **-v** (for verbose) option is generally recommended when using catalina on the Propeller - it makes Catalyst echo each command in the script as it is executed. Given that some compilations can take a long time, this gives a useful indication of how far the compilation has progressed.

For example, here is the output that might be produced by the above **catalina** command if **-v** is *not* included:

```
Catalina Version 8.0

code =     5380 bytes
cnst =      104 bytes
init =      332 bytes
data =        0 bytes
file =    14368 bytes
```

[7] Remember that Catalyst uses **/** and not **** as the path separator - i.e. it is more like Linux than Windows.

And here is the output that might be produced by the above **catalina** command if **-v** *is* included:

```
Catalina Version 8.0
rm  -k /tmp/hello.cpp /tmp/hello.rcc hello.s catalina.s
catalina.cmd /hello.bin
cpp -I/ -I/include -D_POSIX_SOURCE -D__STDC__=1
-D__STRICT_ANSI__ -D__CATALINA__ -Dlibci -D__CATALINA_libci
-D__CATALINA_NATIVE -D__CATALINA_P2 demos/hello.c
/tmp/hello.cpp
rcc -target=catalina_native_p2/catalyst /tmp/hello.cpp
/tmp/hello.rcc
bcc -x11 -L/lib/p2/nmm -tdef -lci -p2 /tmp/hello.rcc -o
catalina.s
pstrip catalina.s
spp -I/ -I/target/p2 -Dlibci -DNATIVE -DP2 /target/p2/nmmdef.t
/hello.s
p2asm -v33 /hello.s
rm  -k /tmp/hello.cpp /tmp/hello.rcc hello.s catalina.s
catalina.cmd
binstats  /hello.bin

code =     5380 bytes
cnst =      104 bytes
init =      332 bytes
data =        0 bytes
file =    14368 bytes
```

Also note that if you include the **-u** (untidy) option in the catalina command, Catalina will not delete any intermediate files, including the *catalina.cmd* file it created to execute each sub-process of the compilation. This means that it is easy to re-do the same compilation again. To do so, just add the **-u** option to any **catalina** command, then to re-execute the same command again enter the following:

```
exec catalina.cmd
```

BCC

The self-hosted version of the Catalina C compiler includes a Binder and Library Manager, now known as **bcc**.

You can demonstrate creating and using a library using **bcc** with the C source files *my_prog.c* and *my_func.c*. First, note that you can compile this program without using a library as follows:

```
catalina my_prog.c my_func.c -lci -v
```

However, you can also choose to put *my_func.c* in a library (we will call it my_lib), and then compile *my_prog.c* using the library. To do so, use the following commands:

```
catalina my_func.c -c
bcc -i -e my_func.s
mkdir lib/my_lib
mv my_func.s lib/my_lib
```

```
mv catalina.idx lib/my_lib
catalina my_prog.c -lci -lmy_lib -v
```

The **bcc** command-line options are the same as the PC version. To see them, try **bcc -?**

P2ASM

The self-hosted version of the Catalina C compiler includes the PASM assembler **p2asm**. Like Catalina, this is only 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.

You can use the **p2asm** command to assemble PASM assembly files. For example, consider the following simple PASM program[8]:

```
CON
LED_PIN = 38           ' Pin 38 is LED on the P2 EDGE
TIME    = 180000000/2  ' 1/2 second @180 Mhz
DAT
   org 0
Loop
   drvnot  #LED_PIN
   waitx   ##TIME      ' Toggle pin every half second
   jmp     #Loop
```

If you use the vi text editor to create this file (e.g. as *pintest.asm*), then you can assemble it using p2asm as follows:

```
p2asm pintest.asm
```

This will produce *pintest.bin*.

The **p2asm** command-line options are the same as the PC version. To see them, try **p2asm -?**

CAT_ENV

The self-hosted version of the Catalina C compiler includes the command **cat_env**. This command does the same job as the **catalyst_env** command does on Windows or Linux. For example, the command

```
cat_env
```

might display:

```
CATALINA_DEFINE   = P2_EDGE SIMPLE VT100 USE_COLOR
CATALINA_INCLUDE  = /include
CATALINA_LIBRARY  = /target
CATALINA_TARGET   = /lib
CATALINA_TEMPDIR  = /tmp
LCCDIR          = [Default]
```

[8] Note that on boards other than the P2_EDGE you may need to change the value of **LED_PIN**. For instance, on the P2_EVAL board you might use pin 56 instead of pin 38.

Propeller 1 Platform-specific Notes

Since there is a wide variety of Propeller 1 platforms, each with different HMI options and different amounts of XMM RAM, this section describes various platform-specific differences in Catalyst support.

C3

By default, C3 programs use the high resolution NTSC TV and keyboard connected to the Propeller. However, you cannot use the TV HMI option when using the FLASH RAM – only the serial options (e.g. TTY or PC).

On the C3, the SPI Flash is used to execute the external demo programs. This means loading programs can take several seconds, during which time there may be no indication that the command is being processed.

Also, the C3 has only 64kb of SPI Ram available, which is not sufficient space to execute some of the larger demo programs – notably the **pcom**/**pint** pascal interpreter, or the **jzip** interpreter.

Here are some example commands you could try:

```
cat SAMPLE.PAS              <---- list a text file
vi CATALYST.TXT             <---- edit a text file
dbasic STARTREK.BAS         <---- run a basic program
mkdir my_dir                <---- make a directory
vi my_dir/my_file.txt       <---- edit a file in a directory
ls my_dir                   <---- list the contents of a directory
rm my_dir/my_file.txt       <---- remove a file from a directory
sst                         <---- play a game of Super Star Trek
dbasic ELIZA.BAS            <---- get some psychiatric help
```

On the C3, it is possible to recompile Catalyst to use a VGA or PC HMI option if desired.

DracBlade

On the DracBlade, all the Catalyst binaries are built to use a High resolution VGA HMI plugin, which uses the display and keyboard connected to the Propeller. However, you cannot use the HiRes VGA HMI option when using the XMM RAM – only the LoRes VGA or serial options (e.g. LORES_VGA, TTY or PC).

Note: After each external command or demo program is run, the screen is cleared. Catalyst will usually ask you to enter a key to continue so that you can read the output of the command.

Here are some example commands you could try:

```
cat SAMPLE.PAS              <---- list a text file
pcom SAMPLE.PAS SAMPLE.P5   <---- compile a pascal program
pint SAMPLE.P%              <---- run the compiled program
vi CATALYST.TXT             <---- edit a text file
dbasic STARTREK.BAS         <---- run a basic program
mkdir my_dir                <---- make a directory
vi my_dir/my_file.txt       <---- edit a file in a directory
```

```
ls my_dir                  <---- list the contents of a directory
rm my_dir/my_file.txt      <---- remove a file from a directory
jzip ZORK3.DAT             <---- play a game of Zork
sst                        <---- play a game of Super Star Trek
dbasic ELIZA.BAS           <---- get some psychiatric help
```

On the DracBlade, it is possible to recompile Catalyst (or some parts of Catalyst) to use a PC HMI option (or a low resolution VGA option). This may be required to run some large applications (such as the Pascal compiler) since the High Resolution VGA driver consumes a large amount of Hub RAM space, which limits the stack space available to other programs.

Hydra

All the binaries in this release (as well as Catalyst itself) are built to use a High resolution NTSC TV and keyboard connected to the Propeller.

Note: After each external command or demo program is run, the screen is cleared. Catalyst will usually ask you to enter a key to continue so that you can read the output of the command.

The Hydra cannot simultaneously use the SD Card and XMM RAM, so while Catalyst itself runs, and the demo programs can be compiled and loaded serially (using **payload**) none of the demo programs can be loaded from SD Card using Catalyst. However, both Catalina LMM programs and normal Spin programs can be loaded and run with Catalyst.

Here are some example commands you could try:

```
cat SAMPLE.PAS             <---- list a text file
vi CATALYST.TXT            <---- edit a text file
mkdir my_dir               <---- make a directory
ls my_dir                  <---- list the contents of a directory
rm my_dir/my_file.txt      <---- remove a file from a directory
```

On the Hydra, it is ***NOT*** possible to recompile Catalyst to use a PC HMI option if the XMM RAM is being used – the HX512 does not allow the serial port to be used at the same time.

Hybrid

All the binaries in this release (as well as Catalyst itself) are built to use a High resolution NTSC TV and keyboard connected to the Propeller.

Note: After each external command or demo program is run, the screen is cleared. Catalyst will usually ask you to enter a key to continue so that you can read the output of the command.

Here are some example commands you could try:

```
cat SAMPLE.PAS             <---- list a text file
pcom SAMPLE.PAS SAMPLE.P5  <---- compile a pascal program
pint SAMPLE.P%             <---- run the compiled program
vi CATALYST.TXT            <---- edit a text file
dbasic STARTREK.BAS        <---- run a basic program
```

```
mkdir my_dir               <---- make a directory
vi my_dir/my_file.txt      <---- edit a file in a directory
ls my_dir                  <---- list the contents of a directory
rm my_dir/my_file.txt      <---- remove a file from a directory
jzip ZORK3.DAT             <---- play a game of Zork
sst                        <---- play a game of Super Star Trek
dbasic ELIZA.BAS           <---- get some psychiatric help
```

On the Hybrid, it is ***NOT*** possible to recompile Catalyst to use a PC HMI option if the XMM RAM is being used – the HX512 does not allow the serial port to be used at the same time.

RamBlade

On the RamBlade, Catalyst is configured to use the PC HMI option, and some programs expect a VT100 compatible PC Terminal emulator (such as **payload** in interactive mode).

Here are some example commands you could try:

```
cat SAMPLE.PAS             <---- list a text file
pcom SAMPLE.PAS SAMPLE.P5  <---- compile a pascal program
pint SAMPLE.P%             <---- run the compiled program
vi CATALYST.TXT            <---- edit a text file
dbasic STARTREK.BAS        <---- run a basic program
mkdir my_dir               <---- make a directory
vi my_dir/my_file.txt      <---- edit a file in a directory
ls my_dir                  <---- list the contents of a directory
rm my_dir/my_file.txt      <---- remove a file from a directory
jzip ZORK3.DAT             <---- play a game of Zork
sst                        <---- play a game of Super Star Trek
dbasic ELIZA.BAS           <---- get some psychiatric help
```

TriBladeProp

On the TriBladeProp, Catalyst is configured to use the PC HMI option, and some programs expect a VT100 compatible PC Terminal emulator (such as **payload** in interactive mode).

Here are some example commands you could try:

```
cat SAMPLE.PAS             <---- list a text file
pcom SAMPLE.PAS SAMPLE.P5  <---- compile a pascal program
pint SAMPLE.P%             <---- run the compiled program
vi CATALYST.TXT            <---- edit a text file
dbasic STARTREK.BAS        <---- run a basic program
mkdir my_dir               <---- make a directory
vi my_dir/my_file.txt      <---- edit a file in a directory
ls my_dir                  <---- list the contents of a directory
rm my_dir/my_file.txt      <---- remove a file from a directory
jzip ZORK3.DAT             <---- play a game of Zork
sst                        <---- play a game of Super Star Trek
dbasic ELIZA.BAS           <---- get some psychiatric help
```

It would be possible to recompile Catalyst to use the local display and keyboard on CPU #1, via a proxy driver from CPU #2 (which has access to the SD card).

SuperQuad and RamPage

The SuperQuad and RamPage are not separate platforms – refer to the notes appropriate to the platform to which they are attached.

However, note that the SuperQuad has no XMM RAM, so only SMALL mode programs can be created. This mode does not provide sufficient RAM space to execute any of the larger demo programs.

Progress Messages on Multi-Prop platforms

On multi-prop platforms (such as the TriBladeProp), it is routine to have to load programs into different CPUs. On platforms where the target CPU supports a directly connected screen, Catalyst can display progress messages on that screen (e.g. on a TV connected to the TriBladeProp CPU 1).

This makes it easy to tell that the program is in fact being loaded correctly - but displaying this information can both slow down the load process slightly, and also reduce the size of programs that can be loaded.

Therefore, this functionality is disabled by default. It can be enabled by defining the **DISPLAY_LOAD** command line symbol when compiling the programs in the *Catalina/utilities* folder.

To enable this functionality, run the **build_all** script (**build_all.bat** under Windows) in the *Catalina/utilities* folder, adding **DISPLAY_LOAD** to the command line. For example:

```
build_all TRIBLADEPROP DISPLAY_LOAD
```

Propeller 2 Platform-specific Notes

Propeller 2 boards fitted with supported PSRAM, such as the P2-EC32MB, or which can use the Parallax HyperFLASH/HyperRAM add-on board can use this as External Memory.

The only Propeller 2 platform specific functionality is that Lua can also support storing and executing program code (only) from suitable PSRAM or HyperRAM on platforms that have it, such as the P2_EDGE or P2_EVAL.

P2_EDGE with PSRAM

If the P2 EDGE is fitted with suitable PSRAM (e.g. it is a P2-EC32MB), it can be used specifically to store Lua program code, which allows larger Lua programs to be executed, at the cost of a slight speed reduction[9]. If the P2_EDGE has HyperRAM fitted, see the next section.

To enable the use of PSRAM specifically for Lua program code storage, specify **ENABLE_PSRAM** to the Catalyst or Lua **build_all** scripts. For example:

```
build_all P2_EDGE SIMPLE VT100 ENABLE_PSRAM
```

Note that using the PSRAM this way is supported only by the Lua execution engine (**luax**) which executes compiled Lua programs, and not for the interactive version (**lua**) that executes text programs or the Multiprocessing version (**mlua** or **mluax**).

So if **ENABLE_PSRAM** is specified, only **luax** will be built by the script - but it will be called **luaxp**, so as not to confuse it with the normal **luax**. This means you may need to build Lua twice – once to build the Lua programs that do not use PSRAM, and then again to build **luaxp** (only) to use PSRAM.

For example, to compile Lua in directory *demos\catalyst\lua-5.4.4,* put the executables in *demos\catalyst\image* you might use commands such as:

```
cd demos\catalyst\lua-5.4.4
build_all P2_EDGE SIMPLE VT100
copy src\*.bin ..\image\
build_all P2_EDGE SIMPLE VT100 ENABLE_PSRAM
copy src\luaxp.bin ..\image
```

Note that specifying **ENABLE_PSRAM** is applicable only when using the Catalyst and Lua build_all scripts and Makefiles – it is not a general Catalina symbol that can be used on the Catalina command-line to enable PSRAM in other cases (which is done via the usual mechanism of linking the program with the psram library – i.e.

---

[9] Note that this is not the same as compiling Lua as either a **SMALL** or **LARGE** XMM program, which uses the PSRAM as general purpose External Memory for both C and Lua code, and (in the case of **LARGE**) can use XMM RAM for both code and data. While it is faster than either of these options, and allows larger code sizes than the version that uses only Hub RAM, there will be less Hub RAM available as data space for Lua programs and so it is also in some ways more limited than the other versions. For more information on using External Memory, see the Catalina Propeller 2 Reference Manual. Note also that you cannot use the PSRAM as both Lua code storage and XMM RAM.

adding **-lpsram** to the **catalina** command, or compiling the programs in **SMALL** or **LARGE** mode).

Finally, note that for small Lua programs, the Hub RAM usage of the PSRAM version may not be much smaller than that of the non-PSRAM version – it may even be larger. This is not only because of the additional PSRAM support code required, it is also because the PSRAM version allocates a fixed amount of Hub RAM on startup to use as a PSRAM cache, and for small Lua programs the cache may be larger than the program being loaded. However, the amount of Hub RAM used for Lua code will never increase beyond the cache size no matter how big the program code gets.

P2_EVAL or P2_EDGE with HyperRAM

If the P2_EDGE or P2_EVAL is fitted with the Parallax HyperFLASH/HyperRAM add-on board, it can be used specifically to store Lua program code, which allows larger Lua programs to be executed, at the cost of a slight speed reduction, in the same way as PSRAM (as described in the previous section).

To enable the use of HyperRAM specifically for Lua program code storage, specify **ENABLE_HYPER** to the Catalyst or Lua **build_all** scripts. For example:

```
build_all P2_EVAL SIMPLE VT100 ENABLE_HYPER
```

Note that using the HyperRAM this way is supported only by the Lua execution engine (**luax**) which executes compiled Lua programs, and not for the interactive version (**lua**) that executes text programs or the Multiprocessing version (**mlua** or **mluax**).

So if **ENABLE_HYPER** is specified, only **luax** will be built by the script - but it will be called **luaxp**, so as not to confuse it with the normal **luax**. This means you may need to build Lua twice – once to build the Lua programs that do not use HyperRAM, and then again to build **luaxp** (only) to use HyperRAM.

For example, to compile Lua in directory *demos\catalyst\lua-5.4.4,* put the executables in *demos\catalyst\image* you might use commands such as:

```
cd demos\catalyst\lua-5.4.4
build_all P2_EVAL SIMPLE VT100
copy src\*.bin ..\image\
build_all P2_EVAL SIMPLE VT100 ENABLE_HYPER
copy src\luaxp.bin ..\image
```

ENABLE_HYPER is otherwise similar to **ENABLE_PSRAM**, described in the previous section.

Catalyst Development

Reporting Bugs

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

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

If you want to help develop Catalyst

Anyone who has ideas or wants to assist in the development of Catalyst should contact Ross Higson at ross@thevastydeep.com

Okay, but why is it called "Catalyst"?

In chemistry a catalyst is a substance that facilitates a chemical reaction, but is not itself consumed. Catalyst is intended to facilitate the use of Catalina on the Propeller, but it does not itself consume any Propeller resources.

Acknowledgments

Dr_Acula, for his work on the auto-execute mode.